# PROGRAM SLICING OF EXPLICITLY PARALLEL PROGRAMS

by

Matthew Bridges

A thesis submitted to the Faculty of the University of Delaware in partial fulfillment of the requirements for the degree of Honors Bachelor of Science in Computer and Information Sciences with Distinction

Spring 2002

# PROGRAM SLICING OF EXPLICITLY PARALLEL PROGRAMS

by

Matthew Bridges

# ACKNOWLEDGEMENTS

First, I wish to thank my family for all their support through the years. Also, my many friends here at the University of Delaware deserve mention for their support. Thank you Pete, Kevin, Matt, Marisa, Allison, Jenny, Natalie, and Lisa. Thank you to everyone in the Hiperspace lab who was there to "support" me, including Ben, Mike, Amy, Ves, Sreedevi, Laura, and Jean. Dr. Lloyd and Dr. Bergman, thank you for taking the time to read this thesis. Lastly, a big thank you to Dr. Pollock for all the hard work she has done to help me get this far.

# TABLE OF CONTENTS

# LIST OF FIGURES

# LIST OF TABLES

## ABSTRACT

The main contribution of this thesis is the development of a technique for static interprocedural slicing of shared memory parallel programs. While static interprocedural slicing for sequential codes is well understood and used in a variety of applications, there are no algorithms yet developed for static interprocedural slicing of shared memory parallel programs. To facilitate the static slicing of parallel programs, a new intermediate program representation, the threaded System Dependence Graph (tSDG), is developed to encompass the parallel and worksharing constructs utilized in OpenMP. The concept of transitive dependence is redefined to include dependences caused by conflict edges in shared memory parallel programs, thus enabling interprocedural slicing of the new program representation. An algorithm for interprocedural slicing over the tSDG representation is presented. The slicing algorithm builds on the algorithms already developed for interprocedural slicing of sequential programs and the algorithms developed for intraprocedural slicing of parallel programs.

# Chapter 1

# INTRODUCTION

Previous research in static program slicing has resulted in successful algorithms and various program representations for slicing most types of sequential programs. Static slicing of parallel programs, however, poses additional challenges, and is still an active area of research.

A *program slice* is defined as the set of instructions in a program which affect or potentially affect the values computed at a certain point in the program. The point of interest in the program is called the *slicing criterion*, and is defined by the pair $\langle p, v \rangle$, where $p$ is a point (instruction) in the program, and $v$ is a subset of variables referenced at $p$[Tip95, Wei84]. This research focuses on the use of static slicing, as opposed to dynamic slicing. A *static* slice is defined as the set of all statements that affect or potentially affect the value of variable $v$ used or defined at $p$ as computed by analyzing the program code[Wei84, HRB97]. *Dynamic* slicing utilizes a specific input data set in addition to the program. Therefore, a dynamic slice shows only the dependences that exist during a single execution of the program with that input[Tip95]. In contrast, a *static* slice shows the dependences that exist for any instance of program execution[Tip95]. To the author's knowledge, this is the first research to address static interprocedural slicing for imperative, shared memory parallel programs, written using the OpenMP standard's explicitly parallel constructs.

OpenMP has become the standard for developing efficient, portable parallel programs for shared memory multiprocessors (e.g., SGI's Origin 3000) [Ope97].

The OpenMP standard introduces the concepts of a parallel region and worksharing constructs that have not previously been addressed in research on program analysis, including slicing. When slicing shared memory parallel programs, the slicer must take into account the possible interactions between access and storage of shared variables by different threads that can potentially execute in parallel. To elucidate these interactions for program slicing, a new program representation is developed, called the threaded System Dependence Graph (tSDG). The tSDG is a fusion of the tPDG[Kri98] and the SDG[HRB97] that allows multi-procedure parallel programs to be analyzed. This thesis presents techniques to construct program slices using the tSDG. Previous work on static slicing of parallel programs has been done by Krinke[Kri98] and Cheng[Che97]. However, Krinke's approach was restricted to slicing with conservative assumptions about the effect of subroutine calls, and Cheng concentrated on slicing of object-oriented parallel programs. The contribution of this research is the development of an *interprocedural* program slicing technique for OpenMP parallel programs that takes into account the effect of subroutine calls within parallel constructs, and thus provides more precise slices than Krinke's approach.

Static interprocedural program slicing has many useful applications, including program debugging, testing, maintenance, integration, automatic parallelization, and complexity measurement. Few sophisticated tools exist to assist programmers in either the debugging or optimization of parallel programs. Optimizing compilers for parallel programs cannot directly utilize optimizations for sequential programs, as shown by Midkiff and Padua[MP90]. Thus, both the compiler and the programmer require different or additional tools and methods to develop parallel codes effectively. Static slicing of parallel codes can assist the programmer in understanding their code and the dependences that exist within it. In addition, the compiler can utilize slicing techniques to help guarantee program correctness; for example, when the program contains different memory consistency models for different variables.

The following chapter gives an overview of program representations, slicing, and parallel programming. Subsequent chapters describe the thesis research on the tSDG, static interprocedural slicing of shared memory parallel programs, and a prototype implementation of the techniques.

# Chapter 2

# BACKGROUND: PROGRAM REPRESENTATION AND SLICING

This chapter provides a background discussion of concepts in the area of program representation. It also includes a discussion of the fundamental concepts and algorithms necessary for the research undertaken. In particular, the chapter includes an overview of sequential program slicing, an introduction to the targeted parallel programming environment, and the motivations for slicing parallel programs.

This chapter describes three representations for sequential programs that are commonly constructed and used for automatic program analysis as part of compiler optimization and software development tools: (in ascending complexity) the control flow graph (CFG), the program dependence graph (PDG)[FOW87], and the system dependence graph (SDG)[HRB97]. Each representation incorporates or utilizes information from previous representations.

To illustrate these representations and later, the algorithms, the C program in figure 2.1 will be used throughout the chapter as the basis for a running example. The linearized textual representation of the program in figure 2.1 illustrates one way of viewing a program. This view, however, is useful only to the programmer, not the compiler or program analysis tools. A transformation of the program from source text to a graph structure is usually performed to facilitate analysis and/or modification of the program by the compiler.

```
1: int main(){
2:   int a = 10, b = 4, c = -1;
3:   for(int i = 0; i < 10;){
4:     work(a, &b);
5:     i = i + 1;
6:   }
7:   b = abs(b);
8:   if(b > 10){
9:     b = 10 + b;
10:   } else{
11:     work(a, &b);
12:   }
13:   printf(``%d : %d : %d'',a,b,c);
14:   return 0;
15: }
```

**Figure 2.1:** Sequential Program for Running Example

## 2.1 Definitions and Terminology

### 2.1.1 Basic Blocks

A *basic block* is a consecutive sequence of instructions that has a single entry and a single exit. In other words, execution of the block can only begin with the first instruction, and it can only leave after execution of the last instruction of the block. There are no jumps into or out of the middle of the block.

For example, the instructions on lines 4 and 5 of procedure `main` could be combined to form a single basic block. However, line 3 could not be part of the same basic block because the test condition of the while loop has two exits.

Basic blocks in this thesis are considered to contain only a single instruction. This allows more accurate slices to be calculated, as dependences between basic blocks become dependences between instructions and not multiple instructions. Thus, instructions are not included which may have been part of the same basic block, but which can not affect the slicing criterion.

### 2.1.2 Graph Concepts

**Graph G:** A set of vertices V and the set of edges E connecting the vertices V, written as G(V,E).

**Vertex:** A node in a graph (e.g., CFG, PDG, SDG, CCFG, tPDG, or tSDG) which represents a program construct. A vertex can represent either an instruction, a basic block, or a control structure (e.g., label, if-then, while, etc.) [Muc97]. This is the basic vertex type in each of the graphical representations; additional vertex types will be defined throughout the chapter.

**Edge:** A relation between two (not necessarily distinct) vertices. For all graphical representations, edges will be directed relationships. For the dependence graph representations, an edge will represent a dependence between two vertices. A dependence of $v_j$ on $v_i$ is written as:

$$v_i \xrightarrow{dep} v_j \tag{2.1}$$

**Path:** A path is a sequence of vertices $v_i, v_{i+1}, ..., v_j$ such that for each consecutive pair of vertices $\langle v_k, v_{k+1}\rangle$, there exists a directed edge from $v_k$ to $v_{k+1}$. The existence of a path from $v_i$ to $v_j$ is written as:

$$v_i \longrightarrow^* v_j = (p = \langle v_i, v_{i+1}, \ldots, v_j\rangle | (\forall k(i \leq k < j))(v_k \longrightarrow v_{k+1})) \tag{2.2}$$

**Predecessors / Successors:** The predecessor set of a vertex $v_i$ (Pred($v_i$)) is the set of all vertices $v_j$ where $v_j \longrightarrow v_i$, while the successor set of a vertex $v_i$ (Succ($v_i$)) is the set of all vertices $v_j$ where $v_i \longrightarrow v_j$ [Muc97].

### 2.1.3 Call Sequence

A *call sequence* is a sequence of procedures $\langle p_1, p_2, \ldots, p_n\rangle$ where $p_i$ contains a call site to $p_{i+1}$.

### 2.1.4 Call Graph

A *call graph* of a program with procedures $p_1, p_2, \ldots, p_n$ is the graph $G = \langle N, S, E, r\rangle$ with vertex set $N = \{p_1, p_2, \ldots, p_n\}$, the set S of call site labels, the set $E \subseteq N \times N \times S$ of labeled edges, and the distinguished entry node $r \in N$ (representing the main program), where for each $e = (p_i, s_k, p_j)$, $s_k$ denotes a call site in $p_i$ from which $p_j$ is called. If there is only one call from procedure $p_i$ to $p_j$, the call site $s_k$ may be omitted and the edge written as $p_i \longrightarrow p_j$[Muc97].

## 2.2 Control Flow Graph (CFG)

The control flow graph (CFG) is a common representation that models a program's possible flow of control[ASU86]. As stated in the definition of a vertex, a vertex represents a basic block (of one instruction). Unique BEGIN and END vertices, which do not represent basic blocks, are added to indicate the entry and exit of the procedure, respectively. The BEGIN node is added as the first node of the procedure and has as its successor the first real basic block. Each possible exit from the procedure has as its successor the EXIT node, making it the final node in the graph[Muc97].

The CFG utilizes control flow edges to describe the program. A control flow edge, $v_i \xrightarrow{cf} v_j$, reflects the potential for program execution to flow from vertex (or basic block) $v_i$ to vertex $v_j$. After a vertex V is executed at runtime, control will be transferred (or flow) along one of V's outgoing control flow edges. If V has more than one outgoing control flow edge, a test must be performed as the last or only instruction in the basic block represented by V. Case statements are handled as if they were mutliple `if` statements. Thus, V cannot have more than two outgoing control flow edges, as the tests performed must be of a boolean nature. If a vertex has two successors, the two edges are considered to be labeled with either T (for true) or F (for false), as appropriate for the outcome of the test. For any vertex V in the CFG, it is assumed that a path exists from the BEGIN vertex to V to the END vertex[FOW87].

**Control Flow Path:** A control flow path from vertex $v_i$ to vertex $v_j$ is a path where the edge from $v_k$ to $v_{k+1}$ must be a control flow edge, written as:

$$v_i \xrightarrow{cf}{}^{*} v_j = (\langle v_i, v_{i+1}, \ldots, v_j \rangle | \exists (v_k \xrightarrow{cf} v_{k+1})(i \le k < j)) \qquad (2.3)$$

An example CFG is shown in figure 2.2 for the code from figure 2.1. All edges shown are control flow edges. Vertex $v_3$ is a vertex with two control flow successors because of the test it contains. The example also shows an example of a `for` (or

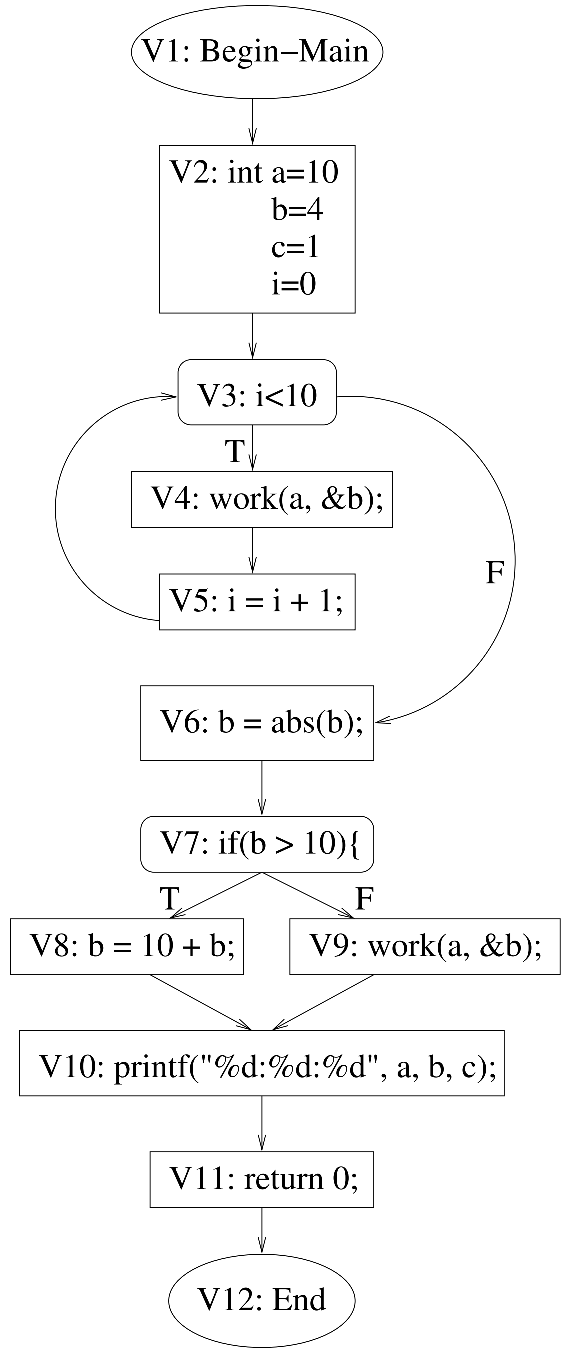


**Figure 2.2:** CFG for Example Program

loop) iteration structure. When the test evaluates to true, control passes into the loop at vertex $v_4$, then to vertex $v_5$ and finally *loops* back to the test vertex $v_3$. When the test evaluates to false, control exits the loop and the program continues at vertex $v_6$. A branching construct is illustrated by the `if` control structure in vertex $v_7$ where control branches to vertex $v_8$ on a true outcome and to vertex $v_9$ on a false outcome. Regardless of the outcome, control passes to $v_{10}$ no matter which path is taken after the `if` executes. There are many control flow paths which exist in the graph, an example of one being the path from vertex $v_6$ to $v_{10}$ as $\langle v_6, v_7, v_8, v_{10} \rangle$. Now that the concept of a CFG has been explained, several additional relationships between vertices of a CFG are defined.

**Dominance:** A vertex $v_i$ dominates a vertex $v_j$ in the CFG iff all control flow paths from BEGIN to $v_j$ pass through $v_i$. Dominance is reflexive (every node dominates itself), transitive (if $v_i$ *dom* $v_j$ and $v_j$ *dom* $v_k$, then $v_i$ *dom* $v_k$), and antisymmetric (if $v_i$ *dom* $v_j$ and $v_j$ *dom* $v_i$, then $v_i = v_j$)[Muc97], and is written:

$$v_i \; dom \; v_j \tag{2.4}$$

**Post-Dominance:** A vertex $v_j$ postdominates a vertex $v_i$ iff all control flow paths from $v_i$ to END pass through $v_j$[Muc97, Tip95], written:

$$v_j \; pdom \; v_i \tag{2.5}$$

**Immediate (Post)Dominance:** A vertex $v_i$ is the immediate (post)dominator of a vertex $v_j$ iff $v_i$ (post)dominates $v_j$ and no other vertex $v_c$ exists such that $v_i$ (post)dominates $v_c$ and $v_c$ (post)dominates $v_j$ [Muc97], written:

$$\begin{aligned} v_i \; (p)idom \; v_j \;\; \equiv \;\; & (v_i \; (p)dom \; v_j) \wedge \neg(\exists v_c)((v_c \neq v_i) \\ & \wedge (v_c \neq v_j) \wedge (v_i \; (p)dom \; v_c) \wedge (v_c \; (p)dom \; v_j)) \end{aligned} \tag{2.6}$$

**Strict (Post)Dominance:** A vertex $v_i$ strictly (post)dominates a vertex $v_j$ iff $v_i$ (post)dominates $v_j$ and $v_i \neq v_j$ [Muc97], written:

$$v_i \; (p)sdom \; v_j \equiv ((v_i \; (p)dom \; v_j) \wedge (v_i \neq v_j)) \tag{2.7}$$

**Dominance Frontier:** A dominance frontier for vertex $v_i$ contains all vertices $v_j$ such that $v_i$ dominates an immediate predecessor of $v_j$, but $v_i$ does not strictly dominate $v_j$[Muc97], written:

$$DF(v_i) \equiv (v_j | (v_j \in V)(\exists v_k \in Pred(v_j))((v_i \; dom \; v_k) \wedge \neg(v_i \; sdom \; v_j)) \tag{2.8}$$

**Reverse Dominance Frontier:** A reverse dominance frontier for $v_i$ contains all vertices $v_j$ such that $v_i$ postdominates an immediate successor of $v_j$, but $v_i$ does not strictly postdominate $v_j$[Muc97], written:

$$RDF(v_i) \equiv (v_j | (v_j \in V)(\exists v_k \in Succ(v_j))((v_i\ pdom\ v_k) \wedge \neg(v_i\ psdom\ v_j)) \tag{2.9}$$

Several examples of each of these properties exist in figure 2.2. Vertex $v_3$ dominates vertices $v_4$ and $v_5$ because flow must pass through $v_3$ from the BEGIN vertex to reach $v_4$ and $v_5$. Vertex $v_3$ also postdominates vertices $v_4$ and $v_5$ because flow must pass through $v_3$ to reach the END vertex. Vertex $v_6$ not only dominates vertex $v_7$ but is also an immediate dominator of $v_7$. The dominance frontier for vertex $v_8$ contains the vertex $v_{10}$, and the reverse dominance frontier for $v_8$ contains $v_7$.

There are several methods for calculating dominance and dominance frontiers based on the structure and features of a language[Tip95, Muc97]. The details of these algorithms is beyond the scope of this thesis. The concepts of (post)dominance and (reverse)dominance frontiers are used for the construction of the program dependence graph.

## 2.3 Program Dependence Graph (PDG)

The program dependence graph (PDG) is another common representation for programs, originally developed by Ferrante, Ottenstein, and Warren[FOW87]. Though its meaning differs depending on the context [Muc97, LC94, HRB97], each formulation is a variation on the original PDG formulation developed by Ferrante, Ottenstein, and Warren[FOW87], intended to make explicit the data and control dependences for each operation in a program. This thesis utilizes the program dependence graph representation as defined by Horowitz *et. al.,* and Lividas and Croll. [HRB97, LC94].

Unlike the CFG which imposes a total execution ordering on the program, the PDG only forces an execution order on those operations dependent on each other.

Thus, the PDG shows only the minimally necessary sequencing of the operations in the program. This property makes the PDG quite useful for automatic parallelization, program slicing, and many optimizations that involve code movement.

For each procedure, a PDG is built that describes data and control dependences of the procedure. As in the CFG, a vertex in the PDG represents a basic block, in addition to BEGIN and END vertices that uniquely indicate entry and exit into and out of the procedure. Unlike the CFG where edges represent potential for execution, edges in the PDG represent data and control dependences. The PDG utilizes the control flow edges of a CFG when determining control dependence edges, thus a CFG is usually built before the PDG is constructed. Figure 2.3 shows the PDG for the example code from figure 2.1.

**Control Dependence Edge (CD):** In a PDG, a directed edge from $v_i$ to $v_j$ written $v_i \xrightarrow{cd} v_j$, that represents the execution dependence of $v_j$ on $v_i$. Control dependence between $v_i$ and $v_j$ is defined in terms of the dominance relation between $v_i$ and $v_j$ and the set of all possible paths P between them.

$v_j$ is control dependent on $v_i$ iff:

1. There exists a control flow path $p \in P$ from $v_i$ to $v_j$, whereby $v_j$ postdominates every vertex in the path $p$, excluding $v_i$ and $v_j$.
2. $v_i$ is not postdominated by vertex $v_j$[FOW87].

$$\begin{aligned} v_i \xrightarrow{cd} v_j \quad \equiv \quad & \neg(v_j \; pdom \; v_i) \wedge (\exists P = \langle v_i, v_{i+1}, \ldots, v_j \rangle) \\ & (\forall v_k, i < k < j \in P)((v_j \; pdom \; v_k)) \end{aligned} \tag{2.10}$$

Essentially, $v_i$ is the closest preceding vertex to $v_j$ whose execution does not guarantee that $v_j$ executes. Thus, there must be at least one other control flow path from $v_i$ to END which does not include $v_j$. This generally occurs from the use of a control structure with a branch, such as an `if` or `while` statement at $v_i$. Thus, every vertex of the PDG is control dependent upon a control structure (if-then, while, etc.) or upon the BEGIN node itself[LC94].

Using the code from figure 2.1 and CFG from figure 2.2, control dependences can be constructed for the PDG of procedure `main`(shown in figure 2.3). Vertex $v_5$ is control dependent upon vertex $v_3$ (the loop test) because there exists a control flow path from $v_3$ to $v_5$, $\langle v_3, v_4, v_5 \rangle$, such that $v_5$ postdominates every vertex in the path except $v_3$ and $v_5$. Thus $v_5$ is control dependent upon $v_3$. Vertex $v_5$ is not control dependent on vertex $v_2$ because the only path from $v_2$ to $v_5$ includes $v_3$, which, as just established, $v_5$ does not postdominate. Thus, there is *not* a control dependence edge from $v_2$ to $v_5$. Having defined control dependence, it remains to define data dependence.

There are four types of data dependence which can occur in the PDG: Def-Use (*flow* dependence), Def-Def (*output* dependence), Use-Def (*antidependence*), and Use-Use (*input* dependence)[Muc97]

**Flow Dependence Edge (Def-Use) (DU):** In a PDG, a data dependence edge from $v_i$ to $v_j$ exists under the following conditions:

1. $v_i$ is a vertex that defines variable $x$
2. $v_j$ is a vertex that uses $x$
3. Control can reach $v_j$ after $v_i$ via an execution path along which there is no intervening definition of $x$. That is, there is a path in the CFG for the program by which the definition of $x$ at $v_i$ reaches the use of $x$ at $v_j$. (Initial definitions (with value "undefined") of variables are considered to occur at the beginning of the CFG; final uses of variables are considered to occur at the end of the CFG.) [HRB97]

It is possible for a vertex to be data dependent upon itself, when the statement resides in a loop.

$$\begin{aligned} v_i \xrightarrow{du} v_j \quad \equiv \quad & (\exists var_a)((var_a \in DEF(v_i)) \wedge (var_a \in USE(v_j)) \wedge \\ & (\exists P = \langle v_i, v_{i+1}, \ldots, v_j \rangle)(\forall v_k, i < k < j \in P) \\ & \neg(var_a \in DEF(v_k))) \end{aligned} \tag{2.11}$$

**Output Dependence Edge (Def-Def):** An output dependence edge exists from vertex $v_i$ to vertex $v_j$ under the following conditions:

1. $v_i$ defines variable $x$
2. $v_j$ defines variable $x$
3. Control can reach $v_j$ after $v_i$ via an execution path along which there is no intervening definition of $x$. [Muc97]

**Antidependence Edge (Use-Def):** An antidependence edge exists from vertex $v_i$ to vertex $v_j$ under the following conditions:

1. $v_i$ is a vertex that uses variable $x$
2. $v_j$ is a vertex that defines $x$
3. Control can reach $v_j$ after $v_i$ via an execution path along which there is no intervening definition of $x$. [Muc97]

**Input Dependence Edge (Use-Use):** An input dependence edge exists from vertex $v_i$ to vertex $v_j$ under the following conditions:

1. $v_i$ and $v_j$ both use the same variable $x$.
2. Control can reach $v_j$ after $v_i$ via an execution path along which there is no intervening definition of $x$. [Muc97]

**Reaching Definition** The definition of a variable $v_i$ is a reaching definition for all uses flow dependent upon $v_i$. [Tip95]

Though there are four kinds of data dependence which can occur between two vertices, only flow dependence is necessary for the slicing algorithm [Tip95]. Horowitz adds output dependence edges to the PDG representation for the purpose of making the PDG *adequate*. That is, if two programs have isomorphic PDGs, they are strongly equivalent[Tip95]. Although output dependence is utilized in other program analysis techniques, it is not necessary for slicing and is not considered henceforth. Also, antidependence and input dependences are not needed because antidependences can be removed by variable renaming, and input dependences play no role in either slicing or optimization because neither statement in an input dependence is changing the value of the variable. Thus, our PDG representation only considers flow data dependences.

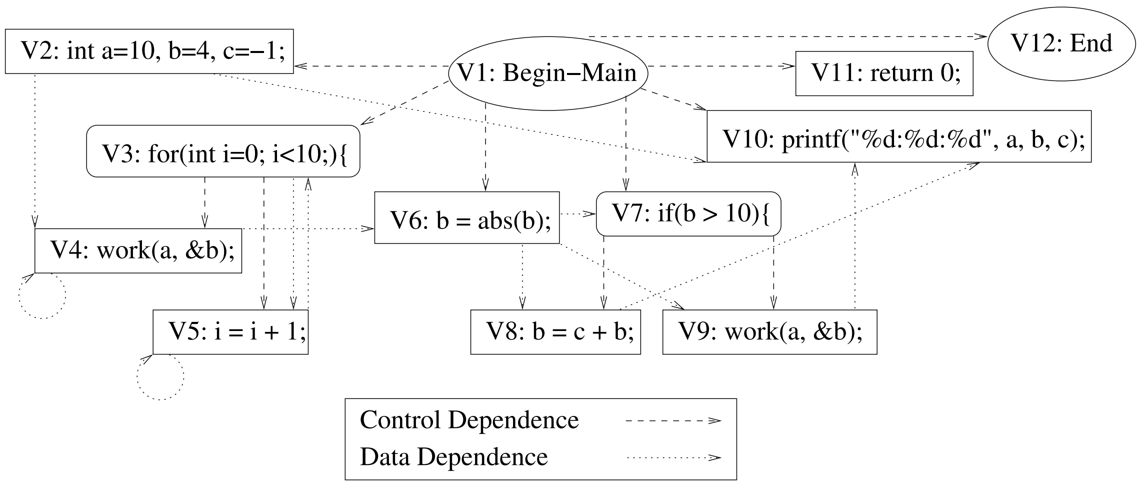


**Figure 2.3:** PDG for CFG in Figure 2.2

An example of def-use data dependence in the PDG for procedure `main` (based on code from figure 2.1 and CFG from figure 2.2) is that of the data dependence between vertices $v_6$ and $v_7$. Vertex $v_6$ defines the variable $b$ which is then used in vertex $v_7$. The necessary control flow path from $v_6$ (the def) to $v_7$ (the use) exists and there is no intervening definition of $b$. Thus, vertex $v_7$ is data dependent upon vertex $v_6$.

## 2.4 Intraprocedural Slicing

Having defined a program representation suitable for individual procedures, formulation of a basic intraprocedural slicing technique is possible. Given a procedure and its PDG representation G, a slicing criterion $\langle p, v\rangle$ of the program is equivalent to slicing its PDG representation on variable $v$, represented as $G/s$, starting at vertex $s$ associated with program point $p$. For single procedure sequential programs, the slice of $G/s$ is a simple graph reachability problem. Vertex $s$ must be directly or transitively control or data dependent on any vertex which affects or potentially affects the value of $v$ at $s$. Thus, if $s$ is transitively control or data dependent upon $v_i$, then $v_i$ is in the set of vertices resulting from slicing $G/s$.

```
**Algorithm IntraSlice**
**Input**: Vertex $s$ to start slicing
        $PDG\ G = (V, E)$
**Output**: The slice S, a subset of the vertices of the PDG G

  worklist $\omega = \{s\}$
  slice $S = \{s\}$
  **repeat**
    remove the next element $x$ from $\omega$
    **for all** edges $(e = (y \xrightarrow{cd,du} x) \in E)$ **do**
      **if** $y$ has not been marked already **then**
        mark $y$ as reached
        $\omega = \omega \cup \{y\}$
        $S = S \cup \{y\}$
  **until** worklist $\omega$ is empty
  **return** $S$
```

**Figure 2.4:** Horowitz, Reps, and Binkley's Intraprocedural Slicing Algorithm using the PDG [HRB97]

All vertices that can be reached via a backwards traversal of the PDG over either control or data dependence edges starting from $s$ are included in the slice for $s$. $V(G/s)$ represents the subset of vertices computed to be in the slice of $s$, shown in equation 2.12 and computed by the algorithm in figure 2.4.

$$V(G/s) = \{w | w \in V(G) \wedge (w \xrightarrow{cd,dd}{}^{*} s)\}\text{[HRB97]} \tag{2.12}$$

An example of a program slice for a slicing criterion $\langle v_9, \{a, b\}\rangle$ for a PDG is shown in figure 2.5, where vertices that are part of the slice are shown in black. Slicing begins at vertex $v_9$ and follows edges as shown below. Upon being added to the worklist a vertex is considered marked as part of the slice.

1. $w = \{v_9\}$

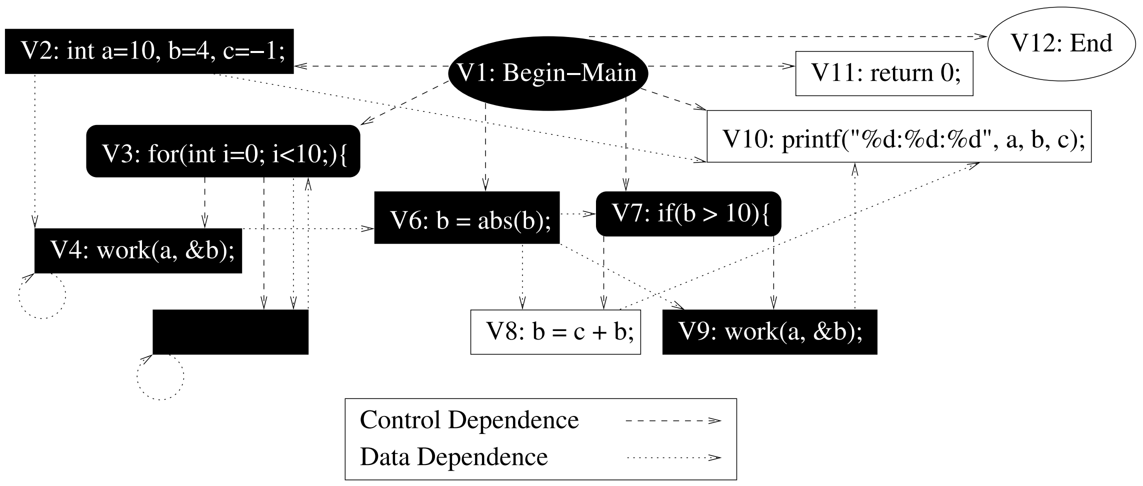


**Figure 2.5:** Intraprocedural Slicing Example

- $v_6 \xrightarrow{du} v_9, S = \{v_9, v_6\}, w = \{v_6\}$
- $v_2 \xrightarrow{du} v_9, S = \{v_9, v_6, v_2\}, w = \{v_6, v_2\}$
- $v_7 \xrightarrow{cd} v_9, S = \{v_9, v_6, v_2, v_7\}, w = \{v_6, v_2, v_7\}$

2. $w = \{v_6, v_2, v_7\}$

   - $v_4 \xrightarrow{du} v_6, S = \{v_9, v_6, v_2, v_7, v_4\}, w = \{v_2, v_7, v_4\}$
   - $v_1 \xrightarrow{cd} v_6, S = \{v_9, v_6, v_2, v_7, v_4, v_1\}, w = \{v_2, v_7, v_4, v_1\}$

3. $w = \{v_2, v_7, v_4, v_1\}$

   - $v_1 \xrightarrow{cd} v_2, S = \{v_9, v_6, v_2, v_7, v_4, v_1\}, w = \{v_7, v_4, v_1\}$

4. $w = \{v_7, v_4, v_1\}$

   - $v_6 \xrightarrow{du} v_7, S = \{v_9, v_6, v_2, v_7, v_4, v_1\}, w = \{v_4, v_1\}$
   - $v_1 \xrightarrow{cd} v_7, S = \{v_9, v_6, v_2, v_7, v_4, v_1\}, w = \{v_4, v_1\}$

5. $w = \{v_4, v_1\}$

   - $v_2 \xrightarrow{du} v_4, S = \{v_9, v_6, v_2, v_7, v_4, v_1\}, w = \{v_1\}$

- $v_4 \xrightarrow{du} v_4, S = \{v_9, v_6, v_2, v_7, v_4, v_1\}, w = \{v_1\}$
- $v_3 \xrightarrow{cd} v_4, S = \{v_9, v_6, v_2, v_7, v_4, v_1, v_3\}, w = \{v_1, v_3\}$

6. $w = \{v_1, v_3\}$

   $v_1$ has no incident edges.

7. $w = \{v_3\}$

   - $v_5 \xrightarrow{du} v_3, S = \{v_9, v_6, v_2, v_7, v_4, v_1, v_3, v_5\}, w = \{v_5\}$
   - $v_1 \xrightarrow{cd} v_5, S = \{v_9, v_6, v_2, v_7, v_4, v_1, v_3, v_5\}, w = \{v_5\}$

8. $w = \{v_5\}$

   - $v_3 \xrightarrow{du} v_5, S = \{v_9, v_6, v_2, v_7, v_4, v_1, v_3, v_5\}, w = \{\}$
   - $v_5 \xrightarrow{du} v_5, S = \{v_9, v_6, v_2, v_7, v_4, v_1, v_3, v_5\}, w = \{\}$
   - $v_3 \xrightarrow{cd} v_5, S = \{v_9, v_6, v_2, v_7, v_4, v_1, v_3, v_5\}, w = \{\}$

9. $Slice = \{v_9, v_6, v_2, v_7, v_4, v_1, v_3, v_5\}$

## 2.5 System Dependence Graph (SDG)

The system dependence graph (SDG) is a program representation introduced by Horowitz, Reps, and Binkley[HRB97] to model dependences between procedures as well as within procedures. The PDG as defined before is designated as the procedure dependence graph, and the SDG links the procedure dependence graphs of the whole program's procedures based on their calling relationships[HRB97]. In this way, interprocedural dependences are represented as well as intraprocedural dependences. The goal of combining these graphs is 3-fold[HRB97].

1. To allow interprocedural slicing in a manner analogous to intraprocedural slicing (i.e., as a graph reachability problem).

2. To build each PDG without any (or with minimal knowledge) of other PDGs.

3. To connect each PDG with other PDGs in an efficient and simple manner.

Extending the representation to reflect dependences of control and data from one procedure to another requires the addition of new variables. Dependences among these new variables allows the computation of dependences across a call site C. Consider a call site C in procedure R with actual parameters $a_1, a_2, ..., a_k$ that calls procedure Q, with formal parameters $f_1, f_2, ..., f_k$. New vertices are defined to model the transfer of parameters between procedures R and Q. At the call site C, two new vertices are created for each parameter $a_1, a_2, ..., a_k$ passed into the procedure: an *actual-in* and an *actual-out* vertex. For procedure Q, two new vertices are created for each formal parameter $f_1, f_2, ..., f_k$ in the procedure header: a *formal-in* and a *formal-out* vertex.

The SDG models the parameter passing using temporary variables and the new vertices just described[HRB97]. Actual parameter $a_i$ of call site C is passed to procedure Q via the intermediate variable $f_i\_in$ for the corresponding formal parameter *f*. Before the call at C is executed, the assignment $f_i\_in \leftarrow a_i$ is made, and after the call at C, but before the the first statement of Q is executed, the assignment $f_i \leftarrow f_i\_in$ is considered to be executed. A different, but analogous variable $f_i\_out$ is used when Q transfers the value of $f_i$ back to $a_i$ in R. When the procedure Q is finished executing, the assignment is made by the assignment $f_i\_out \leftarrow f_i$. Finally, the actual parameter $a_i$ receives the new value by the assignment $a_i \leftarrow f_i\_out$. If the argument $a_i$ is an expression instead of a variable, or a pass by value parameter, then no assignment is made back to the actual parameter $a_i$. To illustrate these new assignments, several new vertices are introduced.

**Call Site Vertex (CS):** Each call site is represented as its own separate vertex in the SDG. Call site C in procedure R that calls procedure Q the $j^{th}$ time is represented by $cs_j^Q$ in the subgraph for the PDG of R [HRB97].

**Actual-In/Out Vertex (AI/O):** These vertices represent parameter transfer from the calling procedure's point of view for each actual parameter $a_i$ and its corresponding formal parameter $f_i$. They represent the instructions $f_i\_in \leftarrow a_i$ for parameter transfer to the called procedure, and the instruction $a_i \leftarrow f_i\_out$ for parameter transfer from the called procedure, respectively. If the argument $a_i$ is an expression and not a variable, then only an actual-in vertex, $f_i\_in \leftarrow a_i$, is constructed, as it is not possible to assign to an expression, obviating the need for an actual-out vertex, $a_i \leftarrow f_i\_out$. Also, actual-out vertices are not constructed for pass by value parameters, as their values cannot be changed, and so should not receive a value back from the called procedure. Each actual-in/out vertex is control dependent upon the call site vertex for the corresponding call-site [HRB97].

$$(\forall a \in (\forall k\ actual\_in_k \cup \forall m\ actual\_out_m)(cs_j^Q \xrightarrow{cd} a) \tag{2.13}$$

**Formal-In/Out Vertex (FI/O):** These vertices represent parameter transfer from the called procedure's (i.e., Q's) perspective. They contain the instructions $f_i \leftarrow f_i\_in$ for parameter transfer into the called procedure's formal parameters, and the instruction $f_i\_out \leftarrow f_i$ for parameter transfer from the called procedure's formal parameters back to the caller's actual parameters. Each formal-in/out vertex is control dependent upon the BEGIN vertex of the called procedure Q [HRB97].

$$(\forall f \in (\forall k(formal\_in_k \cup formal\_out_k))(BEGIN^Q \xrightarrow{cd} f) \tag{2.14}$$

Using this model, data dependences between procedures are limited to dependences from actual-in vertices to formal-in vertices and from formal-out vertices to actual-out vertices [HRB97]. Figure 2.6 shows the original C program of figure 2.1 with the code for the `work` procedure included. Figure 2.7 shows the SDG for this entire program which now includes the vertices just defined. A call site vertex is shown at both vertex $v_4$ and vertex $v_9$. In the SDG, vertices $v_{13}$, $v_{14}$, $v_{16}$, and $v_{17}$ are actual-in vertices, while vertices $v_{15}$ and $v_{18}$ are actual-out vertices. Procedure `main` has no arguments and thus no formal-in or formal-out vertices. Procedure `work`, on the other hand, has two arguments, with vertices $v_6$ and $v_8$ as the formal-in/out vertices for argument $u$ and vertices $v_7$ and $v_9$ as the formal-in/out vertices for argument $v$. To connect the PDGs with the new vertices to form the system dependence graph, three new kinds of edges are added:

**Call Edge (CA):** For each call site C, a call edge is created from the call site vertex at C in R to the BEGIN node of the caller Q. Call edges represent a control dependence of the execution of procedure Q on the execution of the procedure call at C in R[HRB97].

$$(\forall j)(cs_j^Q \xrightarrow{ca} BEGIN^Q) \tag{2.15}$$

**Parameter-In/Out Edges (PI/O):** For each call site C in R, each actual-in/out vertex is linked via a parameter-in/out edge to its corresponding formal-in/out vertex. A parameter-in edge is added from each actual-in vertex of C to the corresponding formal-in vertex in Q. Similarly, a parameter-out edge is added from each formal-out vertex of Q to the corresponding actual-out vertex of C in R, if the actual-out node exists. Parameter edges represent data dependences between procedures[HRB97].

$$(\forall k)actual\_in_k \xrightarrow{pi} formal\_in_k$$
$$(\forall k)formal\_out_k \xrightarrow{po} actual\_out_k$$

Examples of these edge types are also shown in figure 2.7. A call edge exists between vertex $v_4$ of procedure `main` and vertex $v_1$ of procedure `work`. A parameter-in edge exists between $v_{13}$ of `main` and $v_6$ of `work`, while a parameter-out edge exists between $v_9$ of `work` and $v_{18}$ of `main`.

With these new vertices and edges, several PDGs are combined into an SDG, which allows a better analysis of procedure calls when slicing, as the data dependences across procedure calls can be determined. Without the SDG, the slicing algorithm must assume that any pass by reference parameters in a procedure call are altered in the procedure call. That is, if $a_i$ is passed by reference, then, without the SDG, the slicing algorithm would be forced to make the conservative assumption that all actual-in vertices affect the computation of $a_i$.

```c
int main(){
 int a = 10, b = 4, c = -1;
 for(int i = 0; i < 10;){
   work(a, &b);
   i = i + 1;
 }
 b = abs(b);
 if(b > 10){
    b = 10 + b;
  } else{
    work(a, &b);
  }
  printf(``%d : %d : %d'',a,b,c);
  return 0;
}

 void work(int u, int *v){
   *v = *v + u;
   u = 20 - *v;
 }
```

**Figure 2.6:** Example C Program Expanded

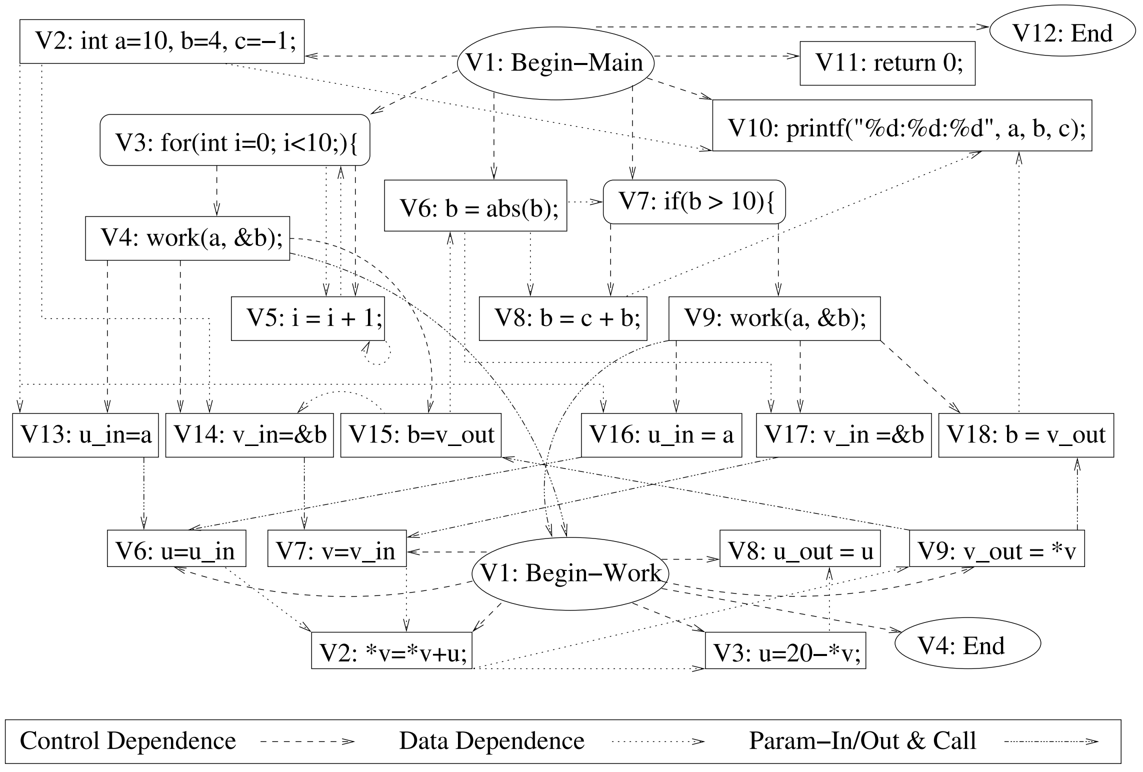


**Figure 2.7:** Example of an SDG

### 2.5.1 Calling Context and Interprocedural Slicing

Once the PDG of the procedures have been linked together at each call site to form a SDG, slicing can be performed on an interprocedural level. However, direct use of the intraprocedural slicing algorithm, phrased as a graph-reachability approach, in figure 2.4, though possible, results in inaccurate slices, due to the calling context problem[HRB97]. The *calling context problem* occurs when slice computation descends into a called procedure P, and ascends out of P to all possible call sites which call P. This allows possibly infeasible execution paths that enter P from one procedure, but exit to a different procedure to be computed. Vertices that are on these infeasible paths may be included in the computed slice, but they are not possibly reached in any execution. This produces inaccurate slices. Further discussion and examples of the calling context problem in relation to slicing are included in section 2.6.

Horowitz, Reps, and Binkley[HRB97] solve the calling context problem and thus compute more accurate slices by extending the SDG representation with transitive dependence edges between actual-in and actual-out vertices. Because of the inclusion of transitive dependence edges, the computation of interprocedural slices is split into two passes over the SDG, discussed in section 2.6.[HRB97]

**Transitive Dependence Edge (TR):** Transitive dependences exist between an actual-in vertex $actual\_in_i$ and actual-out vertex $actual\_out_j$ at a call site C in procedure R when the incoming value in $formal\_in_i$ may be used in obtaining the outgoing value in $formal\_out_j$ of procedure Q[Tip95]. Each $actual\_out_j$ vertex may be transitive dependent upon more than one $actual\_in_i$ vertex. Transitive dependences respresent a data dependence of the actual parameter after the call on zero or more actual parameters before the call. Transitive dependences are calculated between formal variables of Q, but the actual edges themselves are placed at the call site in R.

**Procedure Summary Information:** The set of transitive dependences between formal-in/out variables of a procedure that are reflected back to every call site that calls the procedure[HRB97, Tip95].

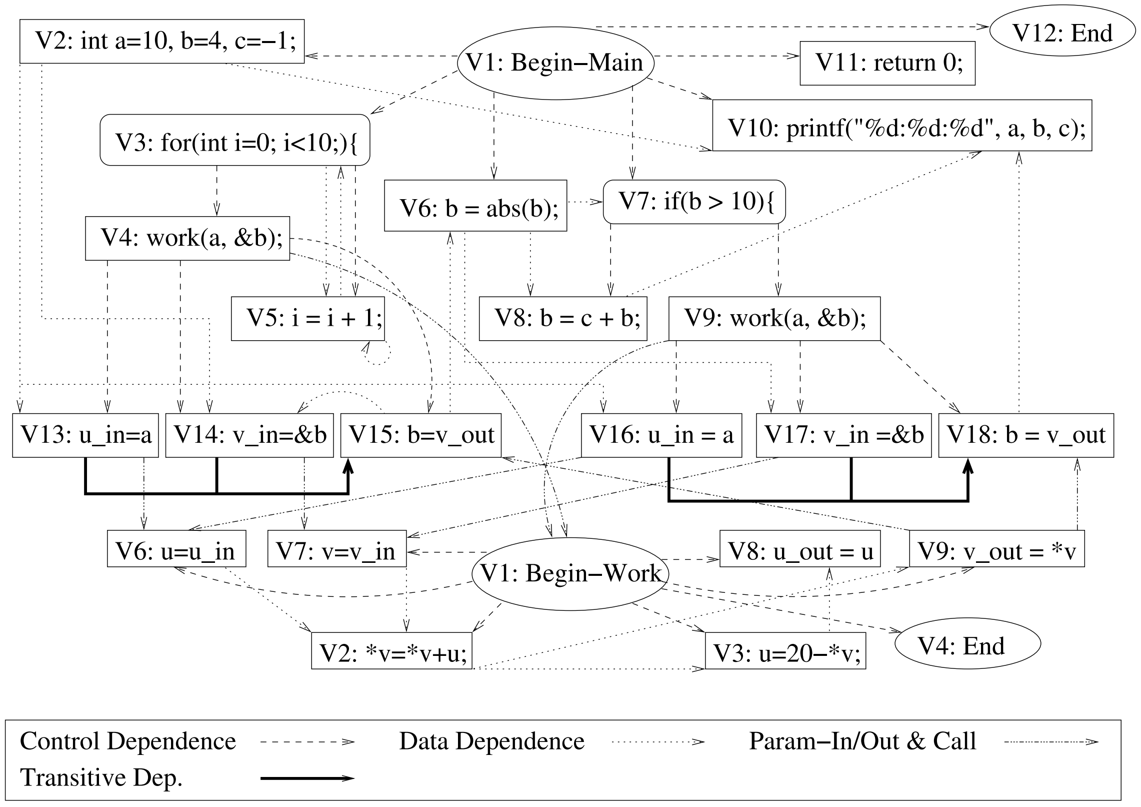


**Figure 2.8:** SDG with Transitive Dependences

Given the SDG in figure 2.7, the formal-in vertex for argument $u$ affects the formal-out vertices for both $u$ and $v$. Exactly how this is deduced is explained in section 2.5.2. An SDG extended with transitive edges is shown in figure 2.8.

Unfortunately, transitive dependences cannot be computed using the transitive closure of a called procedure because the same calling context problem would be encountered. If the language does not allow recursion, then the calling context problem can be avoided by replicating the PDG representation of each procedure at each call site which calls it, forming a *call tree*. This would allow the computation of transitive dependences by determining the procedure summary information for procedures at the leaves of the tree, projecting that information up one level of the tree, and repeating the process until the root of the tree has been reached.

To handle languages with recursion, Horowitz, Reps, and Binkley[HRB97] define a *linkage grammar*. The linkage grammar models the call structure as well as the intraprocedural transitive dependences. Interprocedural transitive dependences are computed using the linkage grammar with a standard attribute grammar construction[HRB97].

An alternative method to determine interprocedural transitive dependences was developed by Lividas and Croll[LC94]; this method does not require the use of a linkage grammar. Lividas and Croll build the transitive dependence edges directly into the SDG while constructing the SDG, instead of after the SDG construction is complete, as Horowitz, et. al., do.

### 2.5.2 Lividas and Croll's Approach to Adding Transitive Dependence Edges

Lividas and Croll form the SDG incrementally, beginning with formation of the PDG for the `main` procedure. Construction branches at call sites to form the PDG for the called procedure. A procedure Q is considered *solved* when all control and data dependences have been computed for the procedure. The procedure

Q is considered *summarized* when all transitive dependences, and thus summary information, have been calculated. Summarizing a procedure Q requires that it first be solved.

For non-recursive programs, Lividas and Croll starts with a call sequence graph initialized to the `main` procedure. A *call sequence graph* is a call sequence where each item in the list is the partial solution to a procedure. When a call site to procedure Q is encountered in the procedure being solved, construction of Q's PDG begins only if a PDG has not already been constructed for Q (Q has not been solved). Assuming the PDG has not already been constructed, there are two cases to consider. First, if Q is *terminal* (i.e., contains no calls to other procedures), then the summary information is computed and reflected back to the call site. Because Q contains no calls, the calling context problem can not be encountered and transitive dependences can be computed for Q's summary information. Second, if Q is not terminal (i.e. contains calls to other procedures), construction on the PDG for Q begins, but must stop at call sites encountered. As before, when a call site is encountered, a partial solution to the PDG is saved, and construction of the called procedure's PDG begins, assuming the called procedure has not already been summarized. This recursive process is guaranteed to stop, since the only ways it could continue indefinitely are if there exists a set of mutually recursive procedures (creating a loop in the call sequence) or if there were infinitely many call sites. Since, the program is not recursive, no loops exist in the call sequence for the program, and since the program is of finite size, there must be a finite number of call sites. Thus, this method guarantees the SDG is built with the proper transitive dependence edges, but only in a non-recursive environment. As a byproduct of using this method, the SDG is built in one pass through the program.

To obtain the transitive dependence edges, Lividas and Croll redefine the way a transitive dependence edge is determined. Also, Lividas and Croll add other edges to fully describe the properties of the called procedure to the caller, as the programs Lividas and Croll consider include the possibility of return values for procedures, which Horowitz, Reps, and Binkley do not.

**Transitive Dependence Edge (TR):** A transitive dependence edge exists from an actual-in vertex $actual\_in_i$ to an actual-out vertex $actual\_out_j$ if the formal-in vertex $formal\_in_i$ is intraslice-path reachable from the formal-out vertex $formal\_out_j$[LC94]. A vertex $v_i$ is *intraslice path reachable* from a vertex $v_j$ if $v_i \in V(G/v_j)$ where G is the procedure's PDG.

$$actual\_in_i \xrightarrow{tr} actual\_out_j \equiv (formal\_in_i \in (G/formal\_out_j)) \tag{2.16}$$

**Return Link Edges (RL):** For each return site in the called procedure Q, a *return-link edge* is created from the return vertex in Q to each call site that calls Q.

$$(\forall p \in P)(\forall cs^Q \in p)(\forall rs \in RS(Q))(rs \xrightarrow{rl} cs^Q) \tag{2.17}$$

Where P is the set of procedures and RS(Q) returns all return vertices for the procedure Q.

**Affect-Return Edges (AR):** If the call site C expects a return value, then the vertex representing the return value of C in the calling procedure is *affect-return* dependent on each actual-in vertex $actual\_in_i$ which corresponds to the actual parameter that influences the returned value and is incident to the procedures' call site vertex[LC94]. Affect-return dependence is essentially a data dependence much like transitive dependence, except that the return value is never passed into the procedure, so transitive dependence is not appropriate for this situation.

$$(\exists v_{return})(actual\_in_i \xrightarrow{ar} v_{return}) \equiv (\exists rs \in RS(Q))(formal\_in_i \in (G/rs)) \tag{2.18}$$

**Return Control Edges (RC):** A return control edge indicates the dependence between the return statement of a procedure Q and other statements following the return statement in Q which will not be executed when the program exits on a return statement. That is, execution of the return statement R precludes execution of all statements return control dependent on R. This edge is necessary only when using a syntax-directed method for determining control dependences[LC94]. This edge is not utilized given our definition of control dependence defined in equation 2.10, as it is not syntax-directed.

**Procedure Summary Information** The union of the three sets of dependence edges for a procedure: transitive dependences, affect-return dependences, and return-link dependences[LC94].

The addition of affect-return and return-link edges to Lividas and Croll's procedure summary information is necessary to consider the effects of procedure calls on return values. Horowitz, et. al., do not allow return statements to return variables, which is the reason their summary information contains only transitive dependence edges[HRB97].

An SDG constructed using Lividas and Croll's method looks exactly the same as one constructed by Horowitz, et. al., except that it contains affect-return and return-link edges. An altered version of the example code is shown in figure 2.9, with its SDG shown in figure 2.10.

The SDG shown in figure 2.10 illustrates transitive dependence edges which are deduced using Lividas intra-slice path reachable criterion. As an example, in procedure `work`, since the formal-in vertex for argument $u$ ($v_6$) is intra-slice path reachable from the formal-out vertex for argument $v$ ($v_9$), a transitive dependence exists between the corresponding actual-in and actual-out vertices at each call site. For the call site at vertex $v_4$ in procedure `main`, this means that there is a transitive dependence edge from vertex $v_{13}$ to vertex $v_{15}$. On the other hand, even though the formal-in vertex for argument $u$ ($v_6$) in procedure `work` is intra-slice path reachable from the formal-out vertex for argument $u$ ($v_8$), a transitive edge is not placed at the call site vertex $v_4$ in procedure `main`. This occurs because parameter $u$ is passed by value, so an actual-out vertex for it was not created at this call site. Since there is no actual-out vertex, a transitive dependence edge can not exist.

Examples of affect-return and return-link edges are also shown in figure 2.10. An affect-return edge exists from vertex $v_4$ in procedure `main` where the variable $c$ expects a return value and is affected by the actual-in vertex corresponding to parameter $u$ ($v_{13}$). Much like the transitive dependence, an affect-return edge exists

```
int main(){
 int a = 10, b = 4, c = -1;
 for(int i = 0; i < 10;){
   c = work(a, &b);
   i = i + 1;
 }
 b = abs(b);
 if(b > 10){
    b = 10 + b;
  } else{
    c = work(a, &b);
  }
  printf(‘‘%d : %d : %d’’,a,b,c);
  return 0;
}

 int work(int u, int *v){
   *v = *v + u;
   return (20 - *v);
 }
```

**Figure 2.9:** Example C Code Expanded with Return Values

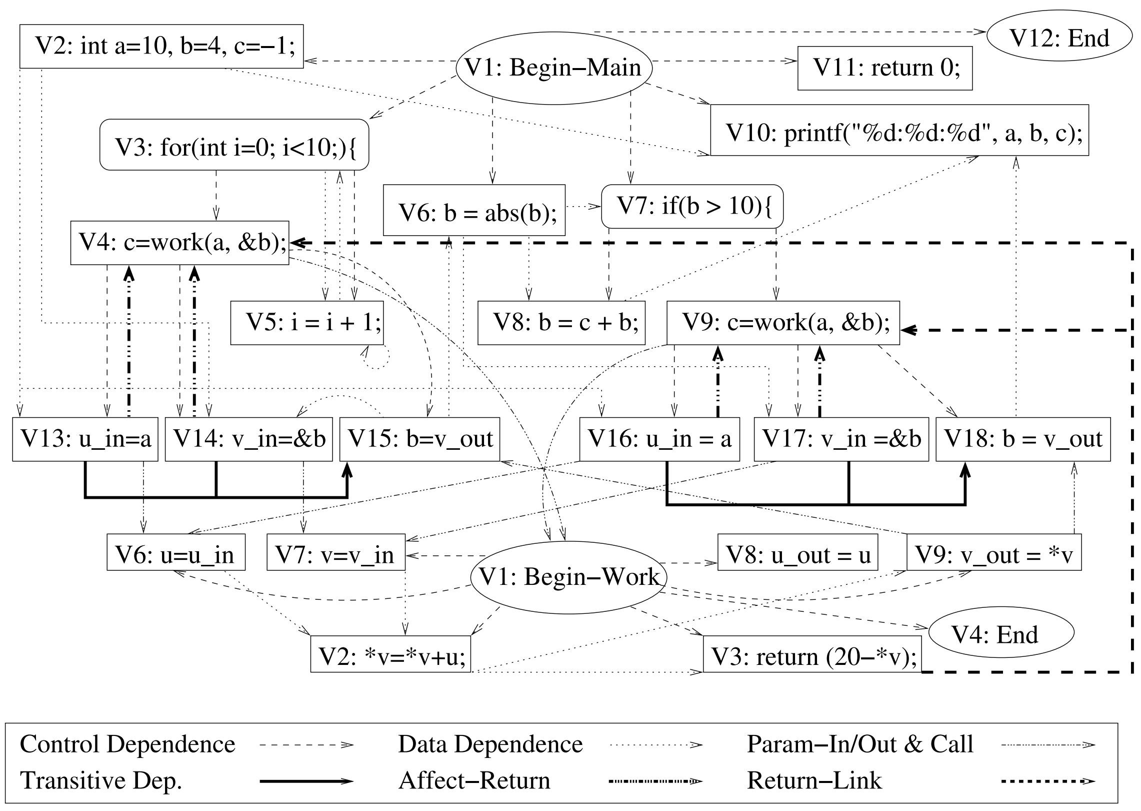


**Figure 2.10:** Example of an SDG with Lividas and Croll's Extensions

between the call-site and the actual-in vertex when the corresponding formal-in vertex ($v_6$) in procedure `work` is intra-slice path reachable from the return vertex ($v_3$) (also in procedure `work`). A return-link edge is illustrated by the edge from the return site in procedure `work` ($v_3$) to a call-site in procedure `main` ($v_4$) that calls procedure `work`.

To handle programs with recursion, Lividas and Croll introduced the *extended call sequence graph* (ECSG) which detects recursive procedures (actually, sets of procedures). The ECSG is a dynamic multilist based on the call sequence graph (CSG) previously defined. The *backbone* of the ECSG is a CSG itself. Associated with each node N in the backbone is a list of procedures referred to as the *iterate* list *rooted* at N. Once a recursive procedure set is detected, the set's members are captured in their own iterate list. Summary information for the entire set is computed iteratively until all procedures in the set have been fully summarized[LC94].

### 2.5.3 Refinement of Actual-Out Vertices

It is possible to further refine the accuracy of slicing by removing actual-out vertices that should not exist. This was partly shown when actual-out vertices were not built when expressions were passed into procedures. In C, actual-out vertices can also be ignored when the parameter is not passed by reference. Since all variables in C not passed by reference are passed by value, actual variables which are not passed by reference are guaranteed not to have their values changed by the called procedure. Since the actual variable cannot be changed, no actual-out vertex needs to be created to reflect the possibility of a new definition for the variable. Finally, even if the parameter is passed by reference, but it is not modified in the called procedure, the actual out node also should not be created for the same reason as a passed by value variable.

Horowitz, Reps, and Binkley[HRB97] use the results of interprocedural dataflow analysis when constructing the procedure dependence graphs to determine which actual-out vertices that can be safely eliminated. The appropriate interprocedural summary information consists of the following sets, which are computed for each procedure P[HRB97]:

**GMOD(P):** The set of variables that might be *modified* by P itself or by a procedure (transitively) called from P.

**GREF(P):** The set of variables that might be *referenced* by P itself or by a procedure (transitively) called from P.

The sets GMOD and GREF are used to determine which parameter vertices to include in PDGs as follows: for each procedure P, the parameter vertices subordinate to P's BEGIN vertex include one formal-in vertex for each variable in $GMOD(P) \cup GREF(P)$ and one formal-out vertex for each variable in GMOD(P). Similarly, for each site at which P is called, the parameter vertices subordinate to the call site vertex include one actual-in vertex for each variable in $GMOD(P) \cup GREF(P)$ and one actual-out vertex for each variable in GMOD(P). (It is necessary to include an actual-in and formal-in vertex for a variable $x$ that is in GMOD(P) and is not in GREF(P) because there may be an execution path through P on which $x$ is *not* modified. In this case, a slice of P with respect to the final value of $x$ must include the initial value of $x$; thus, there must be a formal-in vertex for $x$ in P and a corresponding actual-in vertex at the call to P.)[HRB97]

Lividas and Croll[LC94] present a different method which does not require the calculation of GMOD and GREF sets. Determination of which actual-out vertices to construct can be done with information already contained in the PDG of the called procedure. Since the PDG of each procedure is built as the SDG is built, the proper actual-out vertices can be determined during SDG construction. Lividas and Croll consider four cases:

**Case 1: Never Modified** The variable is passed to the procedure and is *never* modified (i.e., there is no execution path where the variable is defined).

**Case 2: Always Modified** The variable is passed to the procedure and is *always* modified (i.e., there is no execution path where the variable is not defined).

**Case 3: Sometimes Modified** The variable is passed to the procedure and is *sometimes* modified (i.e., there are some paths where the variable is defined and others where it is not).

**Case 4: Unknown** The *initial* condition before the variable has been classified. This case does not exist after the variable has been classified. [LC94]

The second and third cases can be combined for the purposes of slicing, the differentiation being of use to other related applications, such as calculation of reaching definitions[LC94].

## 2.6 Interprocedural Slicing

As in intraprocedural slicing, interprocedural slicing is performed starting at a vertex $s$ in the SDG representation G (in this thesis, an SDG constructed by Lividas and Croll's method). Interprocedural slicing is still fundamentally a graph reachability algorithm but with respect to the SDG. The challenges for interprocedural slicing occur when descending into a called procedure. The context of the call (i.e., the calling procedure) must be maintained. This requires keeping track of the call-order of the procedures during slicing to maintain proper context for ascending back from called procedures. By introducing transitive edges into the SDG representation, the calling context problem is avoided, as the slicing algorithm can step across a call without descending into it[HRB97].

To illustrate the calling context problem, consider figure 2.10. If slicing began at vertex $v_6$ in procedure `main`, it would reach vertex $v_4$, and also descend into procedure `work`. From procedure `work`, it would ascend (via call edges) back to vertex $v_4$, which is correct, but also back to vertex $v_9$, which is incorrect. By including vertex $v_9$, in the slice the accuracy of the slice greatly decreases. This decrease in accuracy is the reason for the addition of transitive dependence edges, which allow the calling context problem to be avoided.

To find the slice $G/s$, two passes are made through the SDG. Since the SDG is based on Lividas and Croll's modifications, the algorithm presented here is Lividas and Croll's version of Horowitz, et. al.'s original interprocedural slicing algorithm[LC94, HRB97]. The first pass of the slicing algorithm finds the set of vertices reachable directly or transitively only by the following edges: control dependence (cd) , def-use dependence (du), parameter-in (pi), transitive dependence (tr), affect-return (ar), and call (ca). Using only these edges, the first pass can only ascend into a procedure; it cannot descend into any called procedure. It is during this pass that the transitive dependence edges described earlier allow the slicer to sidestep the calling context problem. The second pass finds the set of vertices transitively reachable using the edges: control dependence (cd), def-use dependence (du), parameter-out (po), transitive dependence (tr), affect-return (ar), and return-link (rl). This pass can only descend into called procedures. The final slice is the union of these two sets.

For the second pass, it is necessary to keep track of a list of call sites found in the first pass. When slicing during the second pass, the algorithm should also slice on those call sites, so as to include all procedures called by procedures which directly or transitively call the procedure containing $s$. The algorithm for interprocedural slicing of a SDG is shown in figure 2.11 and utilizes the intraprocedural algorithm presented earlier, modified to consider a different subset of edges for each pass.

Applying the slicing algorithm defined in figure 2.11 to vertex $v_6$ in procedure `main` to the SDG in figure 2.10 produces the slice shown in the figure 2.12, where the vertices in the slice are shown in black, after the first pass. As the figure shows, the first pass does not descend into procedure `work`; it uses the transitive edges to step across the procedure call. Figure 2.13 shows the slice produced by the slicing algorithm after the vertices marked by the first and second pass have been combined. The final slice includes vertices from procedure `work` as the second pass was able to

```
Algorithm InterSlice
Input: Vertex $s$ to start slicing
       SDG G = (V, E)
Output: The slice S, as subset of vertices of the SDG

  $S_1 = IntraSlice(s, SDG\ G' = (V, E' = (\forall(cd, du, pi, tr, ar, ca) \in E))$
  $S_2 = IntraSlice(s, SDG\ G' = (V, E' = (\forall(cd, du, po, tr, ar, rl) \in E))$
  $S = S_1 \cup S_2$
  return S
```

**Figure 2.11:** Interprocedural Slicing Algorithm

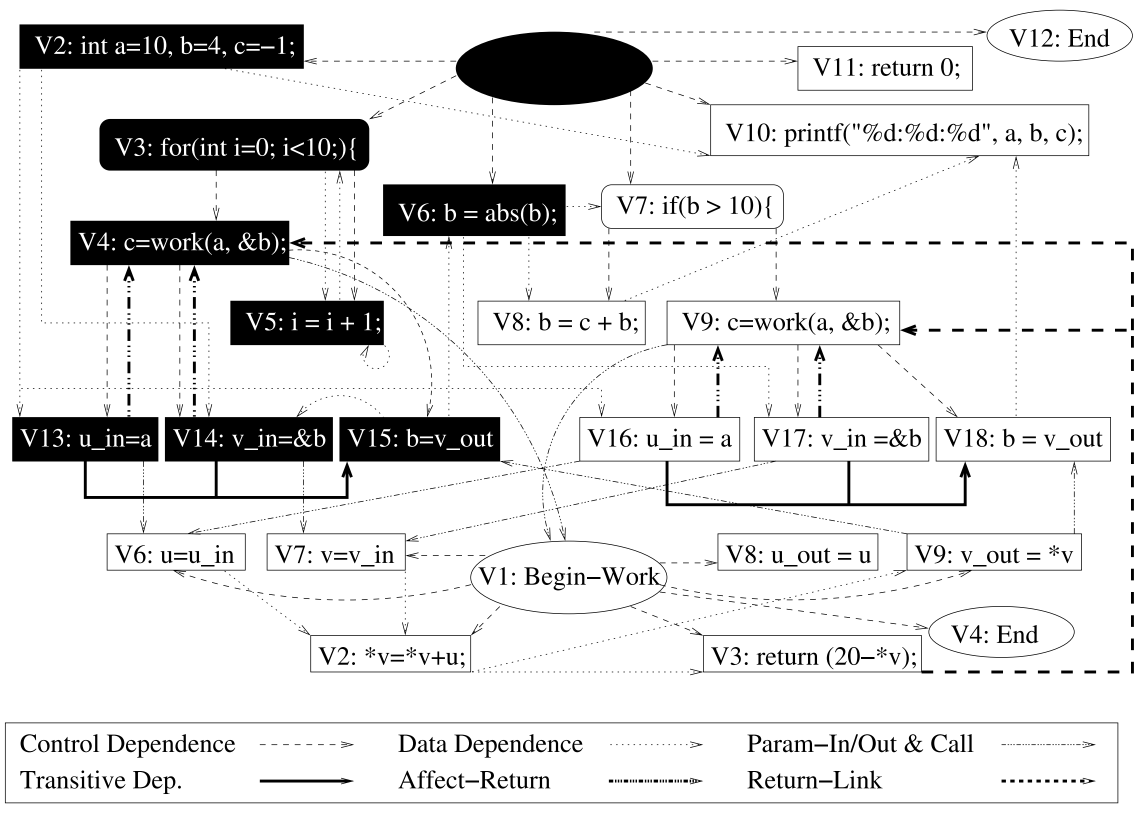


**Figure 2.12:** Interprocedural Slicing Example - Pass 1

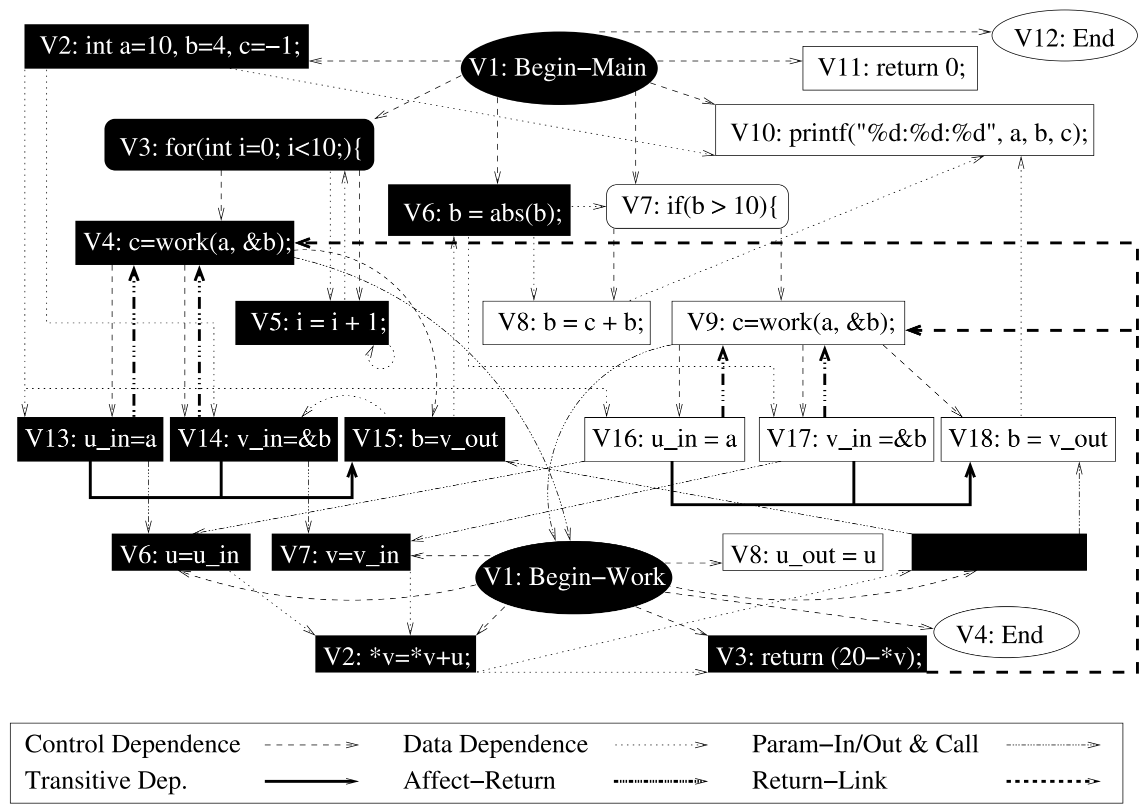


**Figure 2.13:** Interprocedural Slicing Example - Pass 1 and 2

descend into procedure `work` through the parameter-out and return-link edges.

## 2.7 Parallel Programming Environment

There exist numerous libraries and languages for parallel programming. However, two standards have emerged: MPI for the message passing paradigm[SOHL$^+$98] and OpenMP for the shared memory paradigm [Ope97]. This thesis concentrates on the slicing of parallel programs written in the shared memory paradigm due to its preference by programmers and the increasing availability of shared memory multiprocessors, both on large machines and desktop machines. Thus, for practical application of the work, the research has focused on the OpenMP standard. The OpenMP API for Fortran and C/C++ were released in October 1997 and October 1998, respectively[Ope97]. Since then, OpenMP has gained popularity as a standard for developing portable shared memory parallel programs. With the improvements in centralized shared memory technologies and the emergence of distributed shared memory architectures, several physical and logical shared memory configurations are now available. OpenMP builds on these improvements by allowing scalable and portable implementation of parallel programs. The grammar which this research supports is the C language with OpenMP explicitly parallel language constructs. Fortran is also supported through the use of a Fortran to C conversion utility (sf2c).

OpenMP features that represent those most commonly used by programmers for loop-level and SPMD style parallel coding were considered. Loop-level parallelism is the parallelization of looping constructs, usually `for` loops, across multiple threads. In the same vein as loop-level parallelism, SPMD (Single Procedure Multiple Data) works in parallel on different data across different threads, though each thread executes the same code. A mapping of several OpenMP constructs from general parallel constructs is shown in table 2.1.

An explicitly parallel program starts as a single thread of computation, with new threads created when execution of a parallel region begins. There are several

| **Parallel Control Constructs** | |
|---|---|
| Parallel Region | **PARALLEL** |
| Iterative Region | **FOR** |
| Non-Iterative Region | **SECTIONS** |
| Single Process Region | **SINGLE, MASTER** |
| **Data Constructs** | |
| Scope | **PRIVATE, SHARED** |
| Global Object | **THREADPRIVATE** |
| Reduction Construct | **REDUCTION** |
| **Synchronization** | |
| Barrier | **BARRIER** |
| Synchronize | **FLUSH** |
| Critical Section | **CRITICAL** |

**Table 2.1:** OpenMP Constructs

types of parallelism allowed under OpenMP, though, in each, the sequential thread is the master of the parallel threads that it creates.

Task parallelism is achieved through use of the *parallel* directive, and allows multiple threads to execute the same block of code in parallel (i.e., SPMD style). At the end of the block of code, a *barrier* synchronization is implied, and only the master sequential thread continues execution. Task parallelism can be achieved through use of worksharing constructs: the *for*, *sections*, and *single* directives. In a worksharing construct, no new threads are launched, and there is no implied barrier on entry. Loop-level parallelism is accomplished by utilizing the *for* directive inside a parallel region. Iterations of the loop are split across the threads created by the parallel directive in a programmer-controlled manner. The *sections* directive establishes that the specified sections of code are to be divided among the threads and executed once by the assigned thread. Multiple *for* and *sections* directives can be embedded within a *parallel* region. Shortcuts to create parallel *for* and *sections* are given as *parallel for* and *parallel sections* directives. The *single* and *master* directives imply that the associated region shall be executed by only a single thread or the master thread. Other threads in the team wait at the end of the region until execution completes, except when a *nowait* clause is part of the directive. The *master* directive has an implied *nowait* clause; and thus has no barrier on entry or exit[Ope97].

Besides these control constructs, OpenMP provides data constructs for variable scoping. Directives such as *threadprivate*, *private*, or *shared* are used to control the value of a variable that a thread sees. A variable declared *threadprivate* makes the common blocks among threads local to each thread. The *private* construct makes the associated variables private to each member of the thread team. Lastly, *shared* makes the variable shared among all members in the team, thus only one value of the variable exists[Ope97].

Synchronization constructs such as point-to-point, global and atomic synchronization are also provided by OpenMP. Global synchronization is provided by the *barrier* directive, which forces all threads in a team to wait until all threads have reached the barrier before continuing execution. The *critical* directive limits execution of the code to one thread at a time. The *flush* directive requires that each thread has a consistent view of those variables shared between threads. A *flush* directive is implied in the *barrier* directive, at entry to and exit from a critical, and on exit from a *parallel*, *for*, *section*, and end *single*, unless a *nowait* clause is specified[Ope97].

Figure 2.14 illustrates an OpenMP program that utilizes several OpenMP constructs. A *parallel* region begins on line 4 after declaring several variables. Line 5 utilizes the *master* directive to print a line about how many threads were created. Since the *master* directive is utilized, only the master thread will execute the `printf` on line 6. Then, a *sections* directive is started with two *sections* on line 9. In the **first** section, *b* gets a new value on line 11, but since *b* was declared private, only the thread executing the first section block sees the new value. Then, inside the first section, a new parallel region begins on line 13. When entering the new parallel

```
void main(){
int a=10, b=2;
float c[1000];
#pragma omp parallel shared(a,c) private (b)
  #pragma omp master nowait
    {printf(``The master thread has created %d threads\n'',
            (omp_get_num_threads()-1));}

  #pragma omp sections
    #pragma omp section      // first section
    b = a * 1000;

    #pragma omp parallel
      #pragma omp single nowait
      {printf(``Thread #%d got here first\n'',
              omp_get_thread_num());}

    printf(``1:The value of b is %d'', b);

  #pragma omp section        // second section
    #pragma omp parallel
    #pragma omp for schedule(static)
      for(int i=0; i<1000;++i){
        #pragma omp atomic
        c[i] = (c[i] + i) / (b * a);
      }

  printf(``2:The value of b is %d'', b);
}
```

**Figure 2.14:** OpenMP Example Program

region, a new thread team is created. If *nested parallelism* is turned on, then several new threads are created, otherwise only the current thread is included in the thread team. Inside this parallel region, a *single* directive is executed only by the first thread to reach it. Thus, line 15 is executed only once. Finally, the value of $b$ for the first section is printed out at line 18. In the **second** section of the *sections* directive, a new parallel region begins that includes a *parallel for* directive. The *parallel for* directive parallelizes the for loop on line 23 using the specified *schedule* clause. In this case, the schedule is *static* which means that chunks of the for loop are statically assigned to each thread in a round-robin fashion. Lastly, after the parallel for region is finished, the value of $b$ is printed at line 28. The value of $b$ will be different than it was in the first section, because it is not the same copy of $b$.

Utilizing these constructs, many efficient and portable parallel programs can be written for use on shared memory architectures. The language allows pass-by-value and pass-by-reference parameters, pointer operations restricted to pass-by-reference parameters and return statements, and procedures that may return values. In this thesis, extensions for these constructs are left as future work: unstructured control flow(goto, break, continue, etc.), recursion, and aliasing.

## 2.8 Motivations for Slicing Parallel Programs

Parallel programs are more complex than sequential programs. This is due to the fact that parallel programs may have different execution paths when running the same program multiple times with the same data. The non-deterministic nature of parallel program execution makes it harder to debug programs, as an erroneous input may not always produce an error. To combat this increased complexity, program slicing can assist with automatic analysis of a program to highlight an erroneous line and all parts of the program that could affect that line. This allows the programmer to direct their attention only to those parts of the program which can possibly cause an error, reducing the debugging time and effort. Also, slicing of parallel programs can allow the analysis of parallel programs to determine various properties of shared variables. This can be useful when determining such properties as the memory consistency model to use for the variable.

The next chapter identifies the challenges and presents a solution to providing the capability of interprocedural program slicing for OpenMP parallel programs, based on the prior work of interprocedural slicing of sequential programs described in this chapter.

# Chapter 3

# INTERPROCEDURAL STATIC SLICING OF SHARED MEMORY PARALLEL PROGRAMS

## 3.1 Modeling the Control Flow

There are several models of parallelism which can be used to represent parallel programs. The underlying model and environment utilized in this thesis comes from Lee and Novillo [Lee99, NUS98]. An explicitly parallel program starts as a single thread of execution, with new threads logically created when execution encounters a parallel section. For program analysis, threads need only be identified; issues stemming from creation, placement, and scheduling are not considered. This thesis assumes the following environmental characteristics [NUS98]:

1. **Parallelism** Parallel sections are defined using the *cobegin/coend* construct.

2. **Memory Model** Threads run in a shared address space with interleaving semantics (i.e., updates to shared memory made by one thread are immediately visible to other threads). Programs share memory via shared variables. Arrays and aliasing issues are not considered.

3. **Synchronization** Both event-based and mutual exclusion synchronization are supported. Mutual exclusion is used to serialize references to shared variables in the program. We will assume, without loss of generality, that programmers use standard *lock* and *unlock* instructions to serialize across shared variables. Event synchronization is supported using *set* and *wait* instructions.

All the support for event-based synchronization is derived from algorithms in [LMP98].

### 3.1.1 Concurrent Control Flow Graph (CCFG)

Having described the model of parallelism being assumed, a graphical representation for programs can be developed. The concurrent control flow graph (CCFG) for explicitly parallel programs, developed by Novillo[Nov00], is such a model, and is based on the control flow graph (CFG) developed for sequential programs. A CCFG is defined to be a directed graph G(V,E, BEGIN, END) where the set of edges E represents sequential or parallel control flow, conflicts, and synchronization, and the vertices in V are defined essentially as they were in the CFG with a slight modification. Instead of a vertex representing a basic block, it now represents a *concurrent* basic block. BEGIN and END vertices serve exactly the same purpose as they did in the CFG.

**Concurrent Basic Block** A concurrent basic block has the same properties as a basic block with the following additional restrictions[Lee99]:

1. At most one `wait` statement at the beginning of the block.
2. At most one `set` statement at the end of the block.
3. Synchronization operations `lock`, `unlock`, and `barrier` are placed in their own block.
4. Parallel control constructs `cobegin`, `coend`, `parloop`, and `parend` are placed in their own block.

**Parallel Flow Edge** A parallel flow edge is the parallel equivalent of the control flow edge. When control moves from one thread to multiple parallel threads via a COBEGIN, or from multiple parallel threads back to one thread via a COEND, a parallel flow edge indicates the transition from the COBEGIN to the beginning of each thread or from the end of each thread to the COEND[Kri98].

**Conflict Edge** A conflict edge is an edge between two vertices representing blocks that can be executed concurrently and reference the same shared variable. There are two types of conflict edges: def-def (bidirectional), and def-use (one-directional). A def-use conflict edge is analogous to a parallel data dependence edge, where at least one of the references must be a write operation.

**Synchronization Edge** A synchronization edge is an edge that represents an ordering constraint between a `set` and `wait` on the same variable in different threads. Also, synchronization edges can be used to represent mutual exclusion constraints between related `lock` and `unlock` operations.

Synchronization edges and def-def conflicts across concurrent threads are included in the CCFG, but are not used in slicing. However, they are used in other applications, including concurrent static single assignment with mutual exclusion (CSSAME) construction [NUS98].

The CCFG model represents parallel programs as specified by the particular program constructs it allows, which are limited to parallel constructs `cobegin`, `coend`, `parloop`, and `parend` and synchronization constructs `set`, `wait`, `barrier`, `lock`, and `unlock`. This set of program constructs does not correspond to a specific standard. Part of this thesis work has been the modification and extension of the CCFG to represent OpenMP programs.

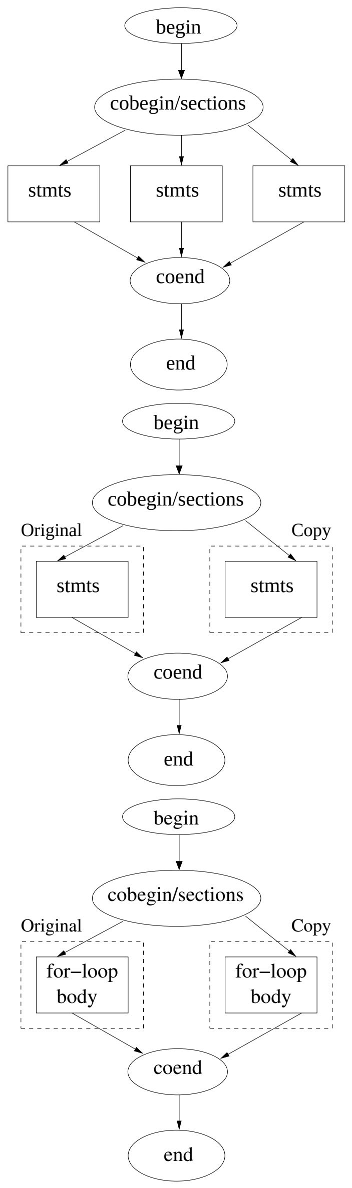
### 3.1.2 Extending the CCFG for OpenMP

In order to limit the modifications to the CCFG representation, a mapping from OpenMP programs to the constructs already represented by the CCFG is provided whenever possible. The combined parallel worksharing constructs `parallel sections` and `parallel for` have direct one-to-one correspondence with the semantics of the `cobegin/coend` and `parloop/parend` constructs, respectively. The OpenMP worksharing constructs `for`, `sections`, `single` can all be embedded within a `parallel` region. The `parallel` region construct is represented as a `cobegin/coend` with two or more threads, where each thread gets a copy of the statement block associated with the parallel region as shown in figure 3.1a. The possibility of more than two threads occurs when a `sections` construct with more than two section bodies is embedded within a `parallel` construct. There must be one thread for each unique section as shown in figure 3.1a. For the cases where no sections are embedded in the parallel region, the body of the original parallel region is replicated as shown in figure 3.1b. In the case of a `for`, the parallel loop is represented by replicating the original body of the loop and considering it to be like a cobegin/coend structure with two or more threads as shown in figure 3.1c. The `for` and `parallel` representations have the drawback of potentially increasing memory requirements, but it is easier to analyze and support than self-referencing conflict edges. Each statement in the copy of the `for` or `parallel` body corresponds to a dual optimization to be applied; it must be applied to both copies. If this is not possible, the optimization is not performed. The `single` and `master` constructs are represented as `cobegin/coend` constructs with one thread. With the available synchronization constructs in the CCFG, the semantics of OpenMP's `barrier`, `flush`, and `critical` synchronization constructs can be simulated easily.

Figure 3.3 represents a CCFG for the OpenMP code from figure 3.2. The OpenMP `parallel sections` directive is utilized to create two separate threads of execution. Conflicts occur between the use of variable $v$ at $v_5$ and $v_6$ with the definition of $v$ in $v_9$ in another thread. Also, the use of $v$ at $v_9$ conflicts with the definitions of $v$ at $v_5$ and $v_6$.

## 3.2 Intraprocedural Slicing of OpenMP Programs

To provide a slicing algorithm for parallel programs, a new representation, the tPDG, was introduced by Krinke [Kri98] (see section 3.2.1). A tPDG is built for each procedure $p$. For each procedure $p$, the set of threads is represented by the set $\Lambda = \langle \lambda_0, \lambda_1, ..., \lambda_n \rangle$. The procedure $\theta(r)$ returns the innermost enclosing thread containing the vertex $r$. $\overline{\Theta}(r)$ is a procedure which returns the set of threads which cannot execute in parallel with the execution of vertex $r$. For convenience $\lambda_0$ is considered to be the main program thread [Kri98]. To assist with the description of the tPDG, a few terms must be defined.



```
#pragma omp sections
{
  #pragma omp section
   { stmts; }
  #pragma omp section
   { stmts; }
  #pragma omp section
   { stmts; }
}
```

```
#pragma omp parallel
  {
   { stmts; }
  }
```

```
#pragma omp parallel
{
 #pragma omp for
  { for-loop
  { for-loop body;}}
}
```

**Figure 3.1:** OpenMP (a)Sections (b)Parallel (c)For Modeled as Cobegin/Coend

```
void work(int u, in *v, int w, int x){
#pragma omp parallel sections shared(v)
#pragma omp section
  if(u>0){
    *v = u + *v;
  }
  else{
    *v = *v + x;
  }
#pragma omp section
  w = *v + w;
  *v = 5 + w;
#pragma omp end parallel
}
```

**Figure 3.2:** Example OpenMP Code Segment

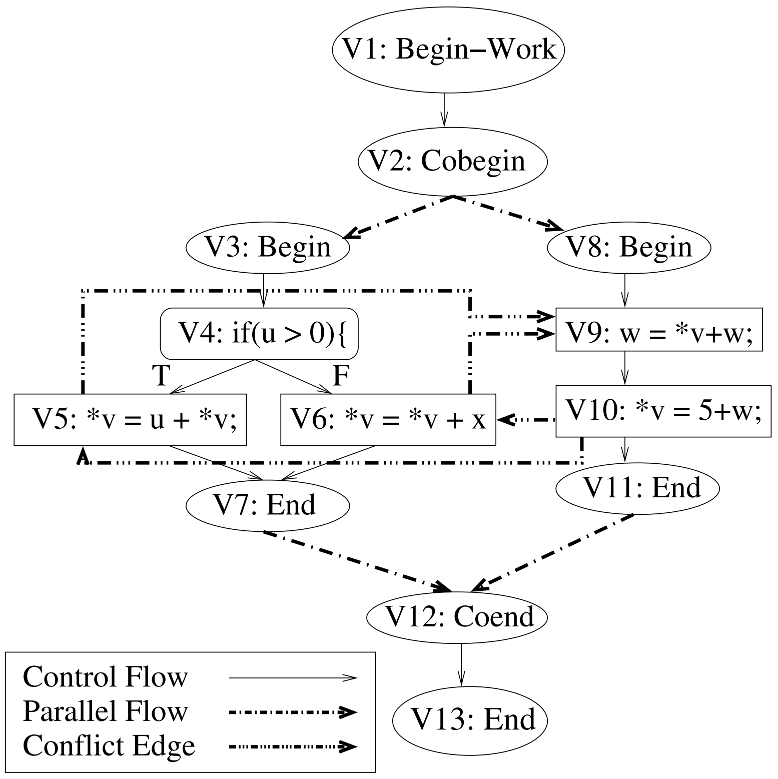


**Figure 3.3:** CCFG for OpenMP Code in Figure 3.2

**Witness:** A sequence of vertices $v_i$ through $v_k$ in a single thread is a *witness* to a possible sequence of execution iff $v_j \overset{cf}{\longrightarrow}{}^* v_{j+1}$ for all $i \leq j < k$. A sequence of vertices is a witness if all vertices of the sequence are part of the path through the CFG in the same order as in the sequence[Kri98]. Every path is by definition a witness of itself.

**Threaded Witness:** A *threaded witness* is an extension of the concept of a witness that incorporates the possible interleaving of vertices by different execution patterns of the threads. A threaded witness is a sequence $\gamma$ of vertices $\langle v_1, v_2, \ldots, v_k \rangle$ in the CCFG such that

$$(\forall t \in T)(\gamma|_t =< m_1, m_2, \ldots, m_j > \Longrightarrow (\forall_{i=1}^{j-1} : m_i \overset{cf,pf}{\longrightarrow}{}^* m_{i+1}) \qquad (3.1)$$

where $\gamma|_t$ is the subsequence of $\gamma$ in which all vertices $v_i$ with $\theta(v_i) \neq t$ have been removed. Essentially this states that a sequence of vertices is a threaded witness iff for every thread, the sequence of vertices belonging to each thread is itself a witness and thus a possible execution pattern. Every ordinary witness in the CCFG is a threaded witness. This definition assures that a sequence of vertices, which are part of different threads, is a witness in each of the different threads, making it a feasible execution pattern[Kri98].

**Interference Dependence Edge (ID):** Though Krinke's interference edges are equivalent to the conflict edges associated with the CCFG, a definition of interference dependence is still given. A vertex $v_j$ is interference dependent on a vertex $v_i$ iff the vertices do not execute inside the same thread, they can be executed concurrently, and $v_i$ contains a definition of a variable that $v_j$ references. Dependences between threads which cannot execute in parallel are ordinary data dependences[Kri98]. Formally,

1. $(\theta(v_i) \neq \theta(v_j)) \wedge (\theta(v_j) \notin \overline{\Theta}(v_i))$
2. $(\exists var)((var \in DEF(v_i)) \wedge (var \in REF(v_j)))$

An example of a threaded witness can be seen by the vertex set $TW = \langle v_2, v_5, v_8, v_{10}, v_{13} \rangle$ in figure 3.3. Since there are three threads in the procedure, $TW$ must be examined with respect to each thread. First, in the sequential thread $\lambda_0$, the subset of $TW$ is $\langle v_2, v_{13} \rangle$. There exists a control/parallel flow path from $v_2$ to $v_{13}$, satisfying the conditions. Second, in thread $\lambda_1$, the subset of vertices is $\langle v_5 \rangle$. Since there is only one node, the condition on the thread is trivially satisfied. Lastly, the thread $\lambda_2$ contains the subset of vertices $\langle v_8, v_{10} \rangle$. Once again, there exists a control/parallel flow path from vertex $v_8$ to vertex $v_{10}$, thus the set $TW$ is a threaded witness. If the vertex $v_6$ were added to make $TW$ equal to $\langle v_2, v_5, v_6, v_8, v_{10}, v_{13} \rangle$, then $TW$ would not be a threaded witness. This would occur because when examining thread $\lambda_1$, there would be no control flow path from $v_5$ to $v_6$. Thus, $TW$ could not be witness to a possible execution of the procedure.

### 3.2.1 Threaded Program Dependence Graph (tPDG)

The threaded program dependence graph (tPDG) was introduced by Krinke [Kri98] as the basis for his static slicing of intraprocedural threaded programs. To construct the tPDG, Krinke introduced the concept of a threaded CFG (tCFG). The tCFG is similar to the CCFG defined in section 3.1, but contains a few differences. The tCFG uses `costart/coexit` instead of `cobegin/coend` to represent statements in parallel sections. Also, the tCFG does not describe how to handle `for` loops, `parallel` regions or synchronization. The tPDG representation constructs control and data dependence edges based on the control/parallel flow and conflict edges in the tCFG using standard algorithms. The *interference* edges utilized by Krinke are identical to the *conflict* edges defined in the CCFG representation. Thus, the CCFG representation is utilized in place of Krinke's tCFG representation in this thesis. Krinke's algorithms for building the tPDG and slicing the tPDG representation are applied to the CCFG.

Dependences introduced by interference between threads using shared variables cannot be handled the same as data dependences between local variables in a single thread (i.e., sequential data dependences) because sequential data dependences are transitive (due to the sequential nature of paths), while interference dependence is not transitive (due to the interleaving of parallel paths). These edges allow the slicing algorithm to ensure that paths containing interference edges during slicing are always threaded witnesses in the CCFG. An example of why interference

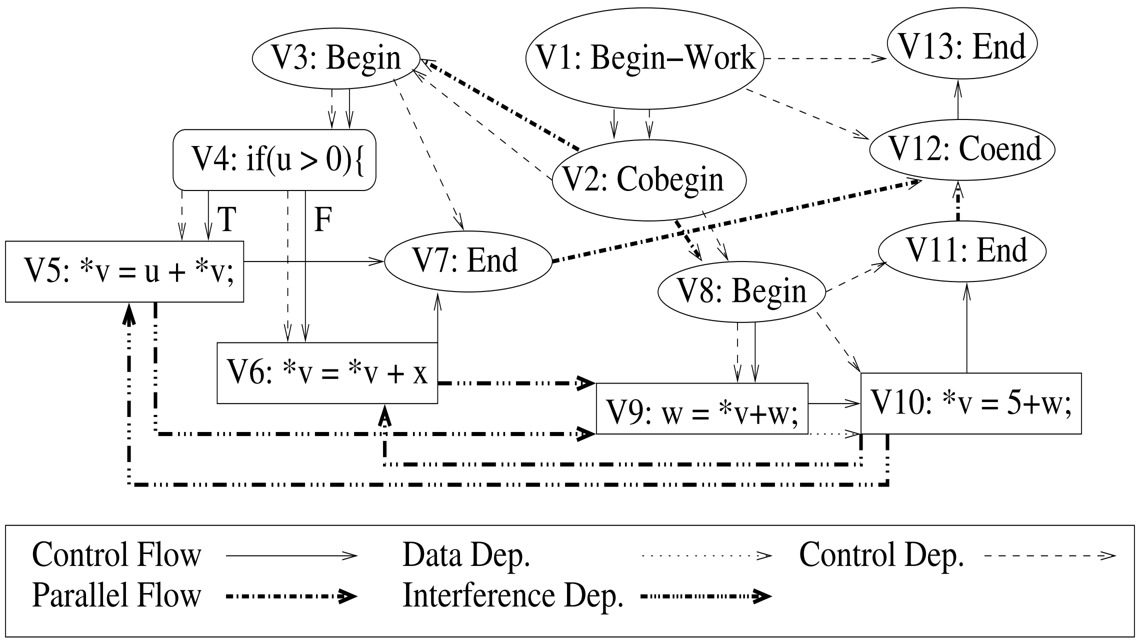


**Figure 3.4:** tPDG for OpenMP Code in Figure 3.2

dependences are not transitive is shown in the slicing example in section 3.2.2. Techniques to calculate interference and synchronization edges are beyond the scope of this thesis, but there exist standard algorithms to calculate them[Kri98].

The tPDG for the OpenMP code given in figure 3.2 and based on the CCFG in figure 3.3, is shown in figure 3.4. In addition, control flow edges are maintained between vertices in the tPDG, unlike in the PDG representation of a sequential program.

### 3.2.2 Algorithm for Intraprocedural Slicing over a tPDG

The tPDG representation is suitable for slicing a single procedure of a parallel program. The slicing algorithm, however, cannot be a simple graph reachability algorithm, as was slicing of sequential programs. The standard slicing algorithm does not apply to slicing the tPDG due to the existence of interference edges, which are not transitive. The slice of a tPDG graph ($G/s$) at a vertex $s$ still consists of the vertices on which $s$ directly or indirectly depends. However, when a vertex $q$ is being tested as part of the slice, $s$ must be more than just transitively dependent upon $q$ via data, control, or interference dependences. The path P from $q$ to $s$ must be a threaded witness. Otherwise, it is possible for vertices which should not be part of the slice to be included in the slice.

$$\begin{aligned} G/s \quad = \quad & (q|(\exists P = \langle v_1, v_2, \ldots, v_k\rangle)((q = v_1 \overset{cd,du,id}{\longrightarrow} \cdots \overset{cd,du,id}{\longrightarrow} v_k = s) \\ & \wedge ThreadedWitness(P))) \end{aligned} \tag{3.2}$$

According to Krinke, using equation 3.2 directly as an algorithm to slice threaded programs is too costly, because the threaded witnesses require a great deal of calculation[Kri98]. Krinke devised another algorithm to calculate intraprocedural slices for threaded programs. Krinke's algorithm uses $n$-tuples $T = (\tau_1, \tau_2, \ldots, \tau_n)$, where $n$ is the number of threads in the procedure, to represent the state of all threads in the procedure (referred to as the *thread state*). Each entry in the $n$-tuple represents an execution state (i.e., current vertex) of a thread in the procedure. At any given time, the value of a thread $\lambda_j$'s state is either $\perp$, which signifies any execution state, or a specific vertex $v_i$ in the tPDG, which signifies that execution of thread $\lambda_j$ has not reached $v_i$ but may still do so.

The purpose of the $n$-tuple is to keep track of the vertex $v_i$ in a thread $\lambda_j$ which was exited when the slicing algorithm followed an interference edge into a different thread, $\lambda_k$. Should the slicing algorithm later follow an interference edge out of another thread $\lambda_m$ back into $\lambda_j$ at vertex $v_j$, and the execution state of $\lambda_j$ has not been set to $\perp$, the slicing algorithm is able to check that $v_i$ is reachable (via transitive control/parallel flow) from $v_j$. In much the same manner that transitive dependence edges eliminate infeasible paths of execution through call sites, use of $n$-tuples eliminates infeasible execution paths over inteference edges. The slicing algorithm ensures that, if the slice contains vertex $v_i$ and vertex $v_j$ is reached as above, that a feasible path of execution from $v_j$ to $v_i$ exists in thread $\lambda_j$. In other words, this ensures that paths over interference edges are always threaded witnesses in the tCFG, and is the reason control and parallel flow edges are kept in the tPDG[Kri98].

The algorithm shown in figure 3.5 uses a worklist of pairs $\langle v, T \rangle$ containing a vertex $v$ and an $n$-tuple $T$ to represent the thread state when $v$ was added to the slice. Each incoming edge to the current vertex $v$ is examined and handled differently based on the type of the edge. If the edge is a control or data dependence from vertex $v_i$ to $v_j$ ($v_i \xrightarrow{cd,du} v_j$), both in thread $\lambda_k$, then the thread state $\tau_k$ is simply updated from $v_j$ to $v_i$. If the two vertices $v_i$ and $v_j$ are from different threads $\lambda_k$ and $\lambda_m$, respectively, then the thread state of $\tau_m$ is changed to $\perp$, and the thread state of $\tau_k$ is updated to $v_i$. Though the dependence is between different threads, the fact that the edge is a control or data dependence means that the two threads can not execute in parallel. Thus, one thread must be able to execute before the other, which allows this case to be treated as a sequential one. If the edge is an interference dependence from vertex $v_i$ to $v_j$ ($v_i \xrightarrow{id} v_j$), in threads $\lambda_m$ and $\lambda_n$ respectively, more calculation is done. If thread $\lambda_m$ has a thread state of $\perp$, then the value of $\tau_m$ is updated to $v_i$ and the value of $\tau_n$ is updated to $\perp$. However, if the value of $\tau_m$ is a vertex $v_k$, then the slicing algorithm checks to see if $v_i$ can possibly be in an execution path with $v_k$. That is, the algorithm checks if there is a threaded witness including $v_i$ and $v_k$, except that the algorithm checks for a control/parallel flow path from $v_i$ to $v_k$, as this costs less than calculating all threaded witnesses[Kri98]. If no control/parallel flow path exists, then the slicing algorithm does not follow the interference edge.

If the program is sequential, it has only one thread, and there can be no edges which cross a thread boundary and thus no interference edges. The algorithm then degenerates into the reachability algorithm of sequential programs.

An example of slicing of a tPDG is shown in figure 3.6. Slicing begins at vertex $v_5$ with thread state $[\perp, v_5, \perp]$. Instead of showing the current status of the slice and worklist after each edge is followed, the pair $\langle v, T \rangle$ (vertex $v$, thread state

```
Algorithm Parallel_IntraSlice
Input: Vertex s to start slicing
    tPDG G = (V, E)
Output: The slice S, a subset of vertices of the tPDG
 $C = (s, (\tau_0, \ldots, \tau_{|\Theta|}))|\tau_i = \begin{cases} s & if\theta(s) = \theta_i \\ \perp & otherwise \end{cases}$
 worklist $\omega = \{C\}$
 slice $S = \{s\}$
 repeat
   remove the next element $c = (x, T)$ from $\omega$
   for all edges $e = y \xrightarrow{cd,dd} x$ do
     $T' = [y/\theta(y)]T$
     if $\theta(y) \neq \theta(x)$ then
       Dependence between threads which cannot execute in parallel
       reset all threads which do not execute parallel to $\theta(y)$
       for all $\tau \in \overline{\Theta}(y)$ do
         $T' = [\perp/\tau]T'$
     $c' = (y, T')$
     if c' has not been calculated already then
       mark c' as calculated
       $\omega = \omega \cup \{c'\}$
       $S = S \cup \{y\}$
   for all edges $e = y \xrightarrow{id} x$ do
     $\tau = T[\theta(y)]$
     if $\tau = \perp$ or $y \xrightarrow{cf,pf^*} \tau \neq y$ then
       Interference edge which can be validly followed
       Update thread $\theta(y)$ and do not change $\theta(x)$
       $c' = (y, [y/\theta(y)]T)$
       if c' has not been calculated already then
         mark c' as calculated
         $\omega = \omega \cup \{c'\}$
         $S = S \cup \{y\}$
 until worklist $\omega$ is empty
 return S
}
```

**Figure 3.5:** Krinke's Threaded Intraprocedural Slicing Algorithm[Kri98]

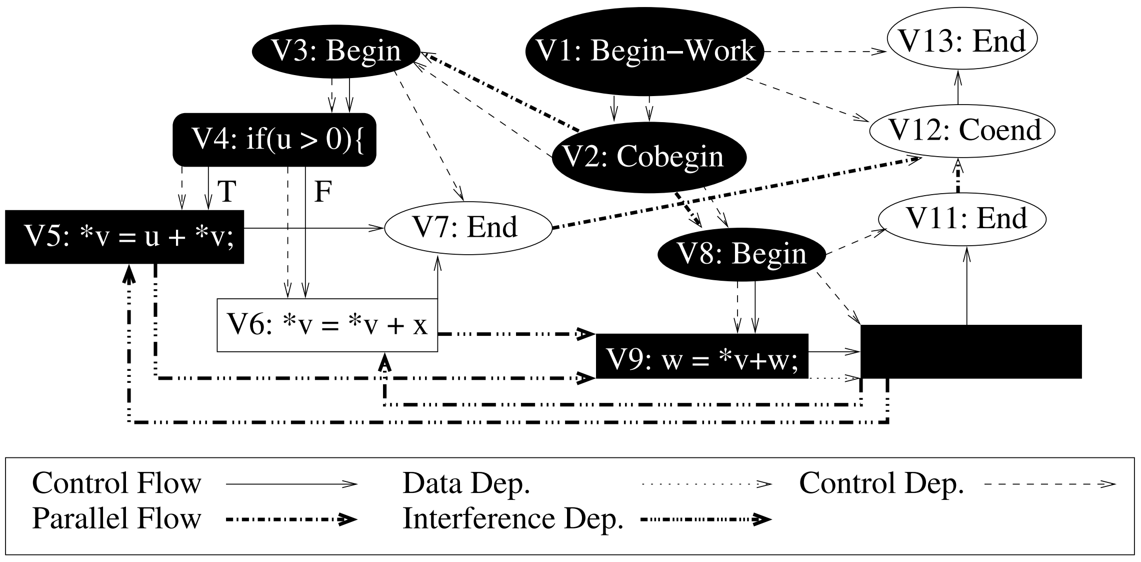


**Figure 3.6:** Threaded Intraprocedural Slicing Example

$T$) which is created by the algorithm after processing the edge is shown. If a pair $\langle v, T\rangle$ is not created, then the reason for this is given instead.

1. $[v_5, [\bot, v_5, \bot]]$

   - $v_4 \xrightarrow{cd} v_5 \triangleright [v_4, [\bot, v_4, \bot]]$
   - $v_{10} \xrightarrow{id} v_5 \triangleright [v_{10}, [\bot, v_5, v_{10}]]$

2. $[v_4, [\bot, v_4, \bot]]$

   - $v_3 \xrightarrow{cd} v_4 \triangleright [v_3, [\bot, v_3, \bot]]$

3. $[v_{10}, [\bot, v_5, v_{10}]]$

   - $v_9 \xrightarrow{du} v_{10} \triangleright [v_9, [\bot, v_5, v_9]]$

4. $[v_3, [\bot, v_3, \bot]]$

   - $v_2 \xrightarrow{cd} v_3 \triangleright [v_2, [v_2, \bot, \bot]]$

5. $[v_9, [\bot, v_5, v_9]]$

- $v_5 \xrightarrow{id} v_9 \triangleright$ *Not Taken:* $\tau_(\theta(v_5)) = v_5$
- $v_6 \xrightarrow{id} v_9 \triangleright$ *Not Taken: No control/parallel flow path from* $v_6$ *to* $v_5$
- $v_8 \xrightarrow{cd} v_9 \triangleright [v_8, [\perp, v_5, v_8]]$

6. $[v_2, [v_2, \perp, \perp]]$

   - $v_1 \xrightarrow{cd} v_2 \triangleright [v_1, [v_1, \perp, \perp]]$

7. $[v_8, [\perp, v_5, v_8]]$

   - $v_2 \xrightarrow{cd} v_8 \triangleright$ *Already Calculated*

8. $[v_1, [v_1, \perp, \perp]]$

   - *No Incident Edges*

9. *Slice* $= \langle v_5, v_4, v_{10}, v_3, v_9, v_2, v_8, v_1 \rangle$

Steps 1 and 5 in the example above illustrate the issue surrounding interference dependence and its lack of transitivity. In step 1 the slicing algorithm followed an interference edge out of thread $\lambda_1$ at vertex $v_5$ into thread $\lambda_2$ at vertex $v_{10}$. In step 5 of the slicing algorithm, an interference edge is followed from vertex $v_9$ in thread $\lambda_2$ back to vertex $v_6$ in thread $\lambda_1$. However, the thread state of thread $\lambda_1$ when following this edge back into thread $\lambda_1$ is $v_5$, so the slicing algorithm must make sure that there exists a control/parallel flow path from $v_6$ to $v_5$, which does not exist. Thus, the edge back to vertex $v_6$ is not followed, given the thread state at that point. If interference dependence had been assumed to be transitive, then $v_6$ would have been incorrectly added to the slice. Vertex $v_6$ cannot possibly affect the value of vertex $v_5$ because they are mutually exclusive when executing. That is, if one executes, the other does not.

## 3.3 Slicing over Call Sites

### 3.3.1 Threaded System Dependence Graph

The threaded system dependence graph (tSDG) is a new representation introduced as part of this work on *interprocedural* slicing of parallel programs. Much like the SDG is an intermediate graph representation for modular programs, the tSDG is an intermediate graph representation for threaded modular programs. A threaded program dependence graph is built for the main procedure, and a threaded procedure dependence graph is built for each auxiliary procedure. Each call-site is then linked as it would be in an SDG for a sequential program. Summary information is generated for each procedure. However, transitive dependence edges between formal-in and formal-out vertices are now computed using Krinke's intraprocedural slicing algorithm[Kri98]. By using Krinke's intraprocedural slicing algorithm as the basis for determining transitive dependences, the effects of shared varaibles can be determined and utilized when building the procedure's summary information. The summary information is used to side-step the calling context problem as described earlier for sequential programs. Summary information consists of the union of the edge sets of transitive dependences, affect-return dependences, and return-link dependences. Naming conflicts will not occur because a static single assignment treatment of all variables is assumed in this work[SHW93].

**Transitive Dependence Edge (TR):** A transitive dependence edge in a tSDG is much like that of an SDG transitive dependence for sequential programs, except that the transitive dependence now utilizes an intraslice-path reachable definition based on Krinke's algorithm which includes the effects of interference dependences from shared variables.

The construction of the tSDG follows the techniques of Lividas and Croll [LC94] in that the tSDG is formed in one pass using summary information reflected back to call sites. It is assumed that a CCFG graph is built for each procedure before tSDG construction is performed on the program. Furthermore, the definition of transitive dependence is modified to use Krinke's intraprocedural slicing algorithm. Since Krinke's algorithm also utilizes interference dependences, in addition to control and data dependence, in determining transitive dependences between formal-in and formal-out vertices, the effects of shared variables inside the procedure can be accurately represented by transitive dependences in the procedure's summary information. By including interference edges in the definition of intraslice-paths, the summary information collected for each procedure captures the shared memory parallel dependences in a tSDG. To compute transitive dependence edges, Krinke's [Kri98] static intraprocedural slicing algorithm, presented in figure 3.5, is utilized.

The key difference between a tSDG and an SDG representation is that the definition of intra-slice path reachable includes the effects of interference edges. The dependences between the shared variable $w$ are reflected to the call site via the transitive dependence edges. If the transitive dependence edges did not take the effects of the interference dependence of shared variables, the tSDG construction would not have placed a transitive dependence edge from the actual-in vertex for parameter $u$ ($v_{11}$) to the actual-out vertex for parameter $w$ ($v_{15}$), as is correct.

### 3.3.2 Algorithm for Interprocedural Slicing over a tSDG

Once the tSDG has been constructed for the program, interprocedural slicing can be performed on the program. Using the techniques developed by Horowitz, et. al., and Lividas and Croll as a basis for slicing of threaded programs, the slicing is done in two passes. This can be done because of the presence of transitive dependence edges, which now include the effects of interference dependence, at call sites. As in sequential slicing, the use of summary information allows the interprocedural slicer to move across a procedure call without decending into it. This prevents descending into a procedure and returning by way of some unrealizable execution path (i.e., side steps the calling context problem). The actual slicing algorithm itself

```
Algoritm Construct_tSDG
Input: CCFG for each procedure in the program P.
Output: threaded System Dependence Graph for P.
1: Initialize call sequence graph (CSG), a linked list of call sites, to main
2: Begin partial solution of main CCFG by initiating computations of
     control/parallel flow, control, data, and interference dependences
3: Upon finding a call to a new procedure, calculation of the dependences of the
     calling procedure is suspended (partial solution preserved); called procedure
     is pushed onto top of CSG
4: if called procedure is already solved, reflect summary information
     of called procedure back onto callee site, pop top of CSG, resume calculation
     of dependencies in calling procedure
5: else call site vertex and entry vertex of called procedure created
6:     for each passed by reference actual parameter, an actual-in node
         is created and an actual-out node is created, also corresponding formal-in
         and formal-out vertices are built
7:     Introduce a call edge from the call site vertex to the corresponding
           procedure entry vertex
8:     for each actual-in node at a call site, introduce a parameter-in
           edge from the actual-in to its corresponding actual-out node
9:     for each formal-out node, introduce a parameter-out edge from
           each formal-out node to its corresponding actual-out node
10:    New dependence calculation is initiated at called procedure,
           including computation of control, data, and interference dependences in
           the called procedure. If formal-out is never modified it is marked as such.
           and the corresponding actual-out node at call site is deleted.
11:    Compute transitive dependence edges (edges from actual-in vertices to
           actual-out vertices) by determining the formal-in vertices that are
           intraslice-path reachable from each formal-out vertex using Krinke's
           static interprocedural slicing algorithm[Kri98].
12:    Compute affect-return edge (edges from actual-in vertices to the return
           at the call site) by determining the formal-in vertices that are
           intraslice-path reachable from each formal-out vertex using Krinke's
           static interprocedural slicing algorithm[Kri98].
13:    Add all return vertices to a return vertex list for use in return-link
           dependences.
14:    Reflect summary information (transitive, affect-return, and
           return-link dependences) of called procedure back onto callee site;
           pop top of CSG.
15:endif
16: resume calculation of dependences in calling procedure (current top of CSG),
     until CSG becomes empty
```

**Figure 3.7:** Algorithm for Construction of tSDG

```
int main(
  int a, b, c, d;
  a = 10;
  b = 6;
  c = 12;
  d = 23;
  work(a, b, &c, d);
  b = c / 2;
  printf(‘‘%d : %d : %d’’,a,b,c);
  return 0;
}

void work(int u, in v, int *w, int x){
#pragma omp parallel sections shared(w)
#pragma omp section
  if(u>0){
    v = u + v;
  }
  else{
    v = v * v;
  }
#pragma omp section
  *w = v + *w;
  x = x + *w;
#pragma omp end parallel
}
```

**Figure 3.8:** Remodeled Example OpenMP Program

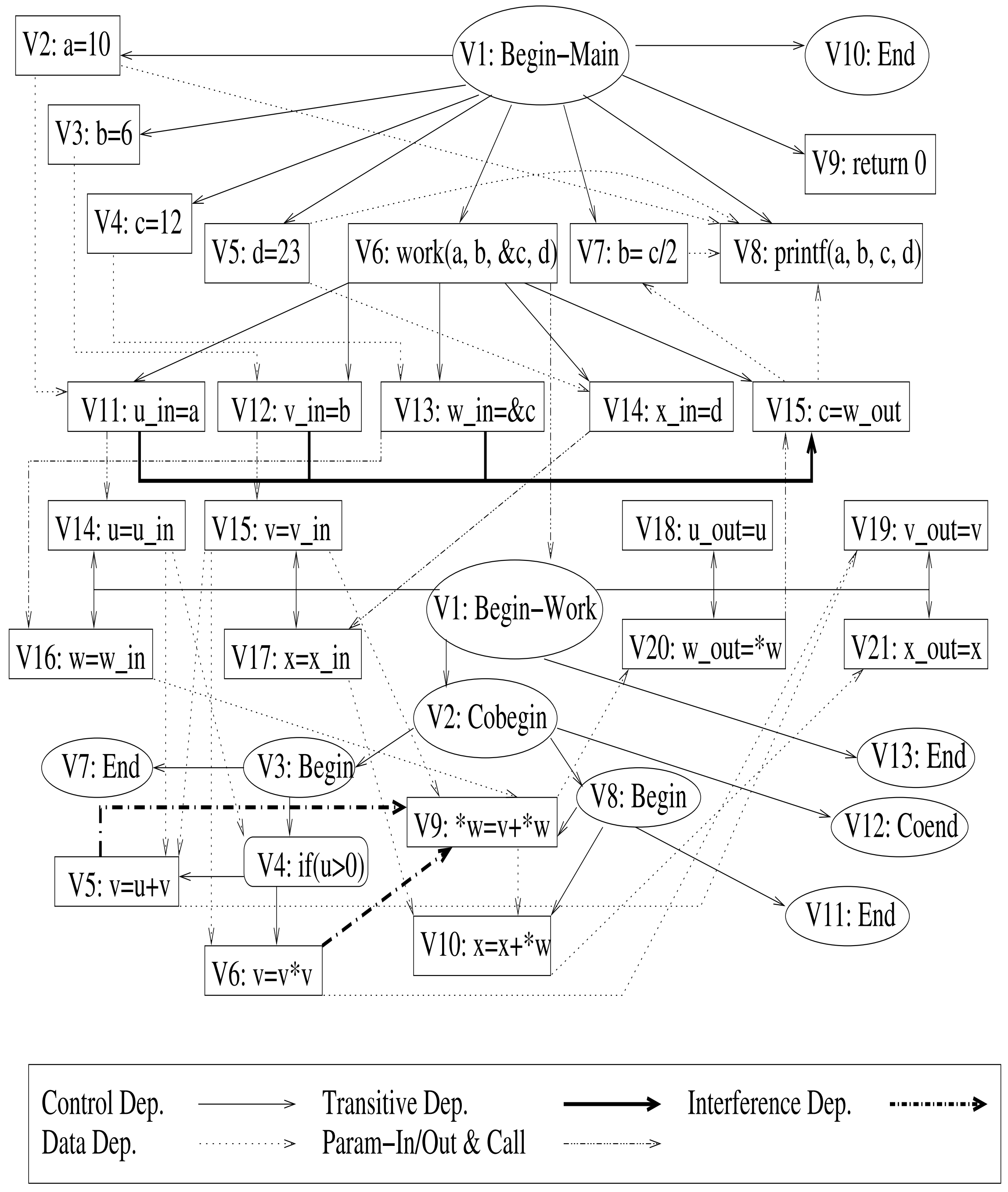


**Figure 3.9:** Example of an tSDG

utilizes the algoritm provided by Krinke[Kri98], modified to deal with interprocedural edges. When the slicing algorithm encounters an interprocedural edge, it simply restarts the intraprocedural slicer at the proper vertex in the newly encountered procedure. The slicing problem is still essentially a graph reachability problem; the slicing algorithm needs to utilize different size $n$-tuples for each procedure.

The first pass marks vertices that reach the slicing criterion vertex $s$ in procedure P, and are thus in the procedure P itself or a procedure that calls P directly or transitively. Starting at the vertex $s$, the slicer ascends on a certain set of edge types: control dependence (cd), data dependence (du), interference dependence (id), parameter-in (pi), transitive dependence (tr), affect-return (ar), and call (ca). During the first pass, the slicer never descends into a called procedure; it only ascends. Through the transitive dependence edges that connect actual-out vertices with actual-in vertices, the slicer is able to determine the effects of a procedure call without descending into it.

The second pass is executed in the same manner as the first pass, but instead of ascending into procedures, it descends into called procedures. Thus, the second pass marks all vertices which reach vertex $s$ from procedures directly or transitvely called by P, or called directly or transitvely by a procedure which directly or transitvely calls P. As in the sequential algorithm, the first pass must keep track of all call sites encountered as the slicing algorithm ascended, so that it can descend into the procedure at each call site. During the second pass, the following edge types are considered: control dependence (cd), data dependence (du), interference dependence (id), parameter-out (po), transitive dependence (tr), affect-return (ar), and return-link (rl). The second pass is complementary to the first in that it descends into call sites that the first pass skipped. As a last step, the set of vertices from each pass are merged together to get the final set of vertices formed by the slice. Slicing can also be performed in the presence of unknown procedures, such as system calls, assuming transitive dependences are known for the procedures.

Using the tSDG example shown in figure 3.9, the first pass of the interprocedural slicing algorithm on the example program from figure 3.8 is shown in figure 3.10. Starting at vertex $v_7$ in procedure `main`, the interprocedural algorithm determines all vertices which are can affect vertex $v_7$ directly or indirectly. Because the tSDG construction includes the effects of interference edges due to shared variables, the transitive dependence edges accurately reflect the dependences the slicing algorithm uses to determine which vertices are part of the slice. The vertices added in the second pass are shown in figure 3.11. In the second pass, the slicing algorithm descends into procedure `work`, and starts a new instance of the intraprocedural slicing algorithm on the vertex descended into (in this case $v_{20}$ in procedure `work`).

As the example illustrates, the formal-out vertex for parameter $w$ is intra-slice path reachable from all formal-in vertices. If interference edges had not been included in the definition of intra-slice path reachable, then only the formal-in vertices for parameters $v$ and $w$ would have been found. Thus, the inclusion of interference edges for shared variables allows the correct deduction of summary information for procedure `work`.

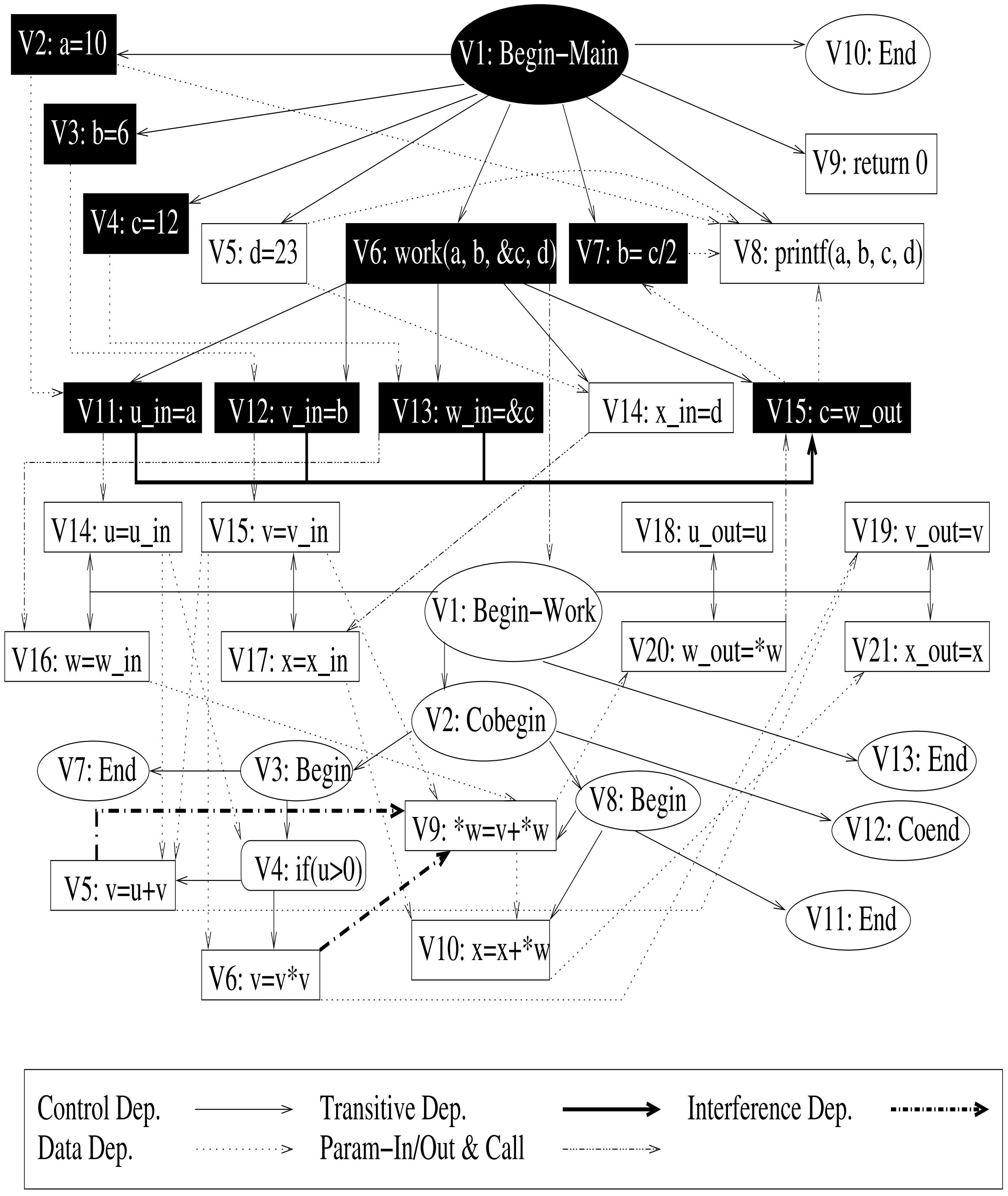


**Figure 3.10:** Slice of tSDG from Figure 3.9 - Pass 1

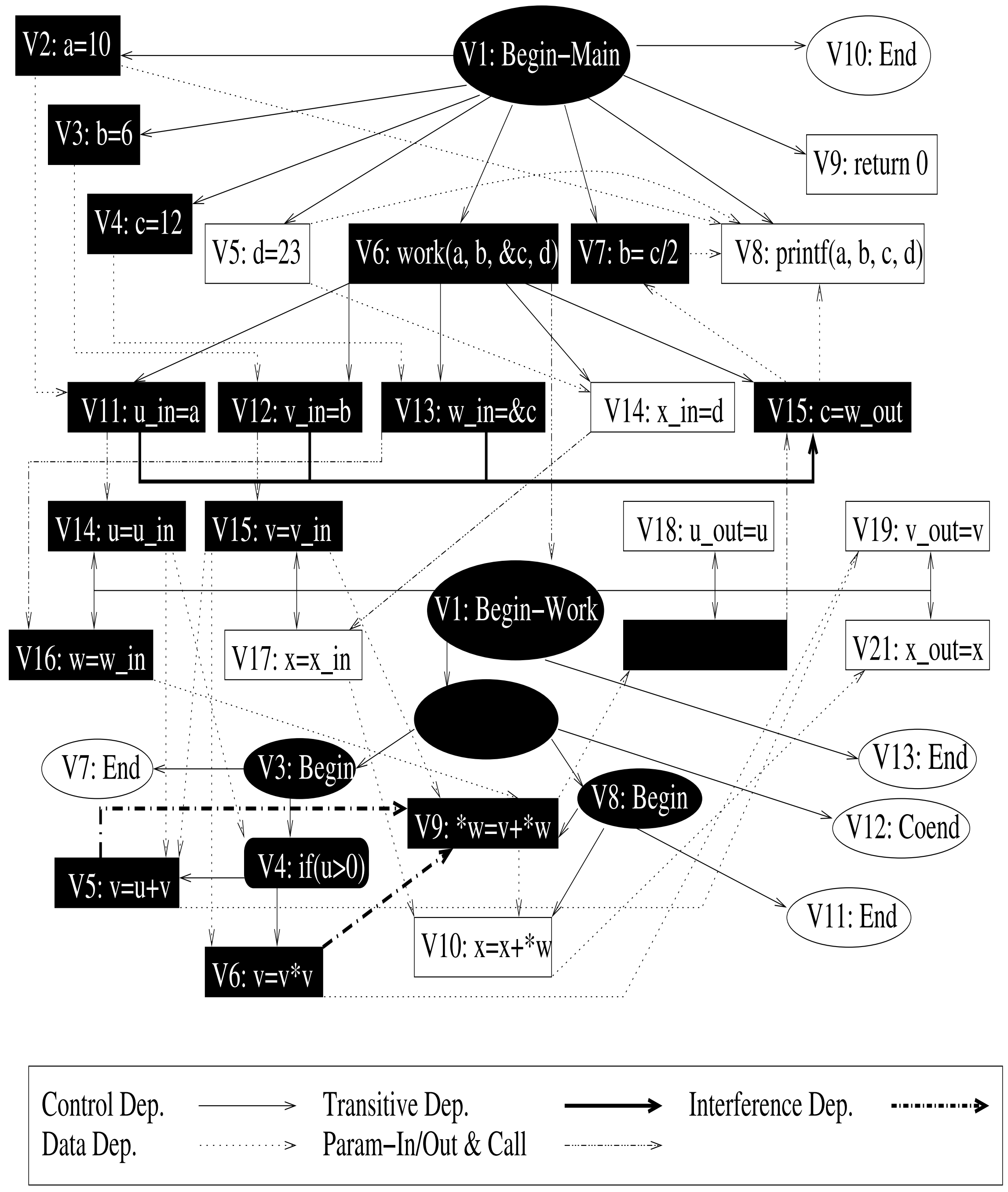


**Figure 3.11:** Slice of tSDG from Figure 3.9 - Pass 1 and 2

# Chapter 4

# IMPLEMENTATION AND EVALUATION

The algorithm presented in the previous chapter for construction of a tSDG was implemented using the SUIF/Odyssey compiler infrastructure. The version of SUIF used was 1.0, which was developed at Stanford University [Gro94]. In addition, the Concurrent Static Single Assignment with Mutual Exclusion (CSSAME) library from the Odyssey framework was utilized. Odyssey was developed by Lee and Novillo [Lee99, Nov00] for the purposes of developing new optimization techniques to take advantage of parallelization and synchronization structures in parallel programs. Lee and Novillo were also interested in adapting traditional optimization techniques to work on explicitly parallel programs. Odyssey builds on SUIF, but bypasses the automatic parallelization that SUIF normally performs. Odyssey utilizes SUIF as a front end to parse the C code, and as a back end to generate a MIPS executable. The overall Odyssey framework is shown in figure 4.1.

The algorithm that builds the tSDG starts with a *.od1* file. This file is produced after SUIF has run its initial parsing algorithms, and Odyssey has completed its identification of the parallel structures. Odyssey currently recognizes only `cobegin/coend` and `parloop` parallel structures. As part of this thesis, a new pass for tSDG construction was developed and integrated between the *FindPar* and *Optimization* phases. The output of the tSDG construction phase is a *.dot* file, which can be shown as a viewable graph via the `dot` program.

The slicing algorithm operates on the tSDG constructed from the *.od1* file. It takes as input a tSDG and currently outputs a *.dot* file which highlights those

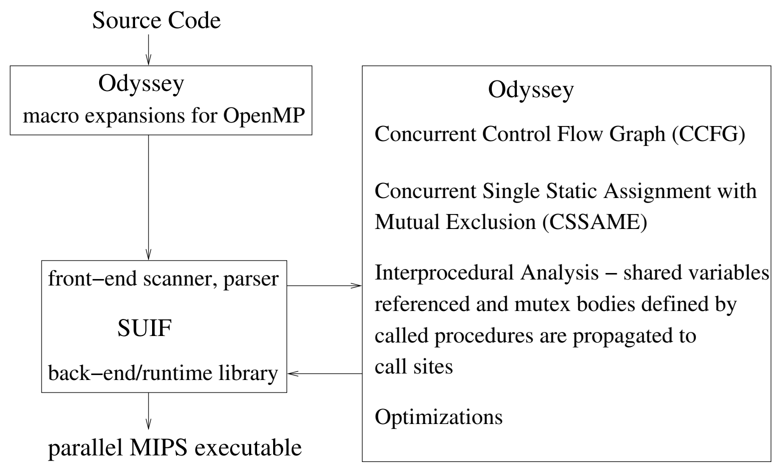

Source Code
Odyssey
macro expansions for OpenMP
Odyssey
Concurrent Control Flow Graph (CCFG)
Concurrent Single Static Assignment with Mutual Exclusion (CSSAME)
Interprocedural Analysis – shared variables referenced and mutex bodies defined by called procedures are propagated to call sites
Optimizations
front–end scanner, parser
SUIF
back–end/runtime library
parallel MIPS executable

**Figure 4.1:** Overview of Odyssey Research Compiler

vertices that are part of the slice. The implementation of the slicing algorithm is not yet complete, but once it has been completed, an evaluation of its efficiency and effectiveness is planned. The implementation will be tested for its accuracy in slicing and the speed with which it slices a program. Also, the slicing algorithm will be used to measure various properties of a program such as usage and interaction of private and shared variables.

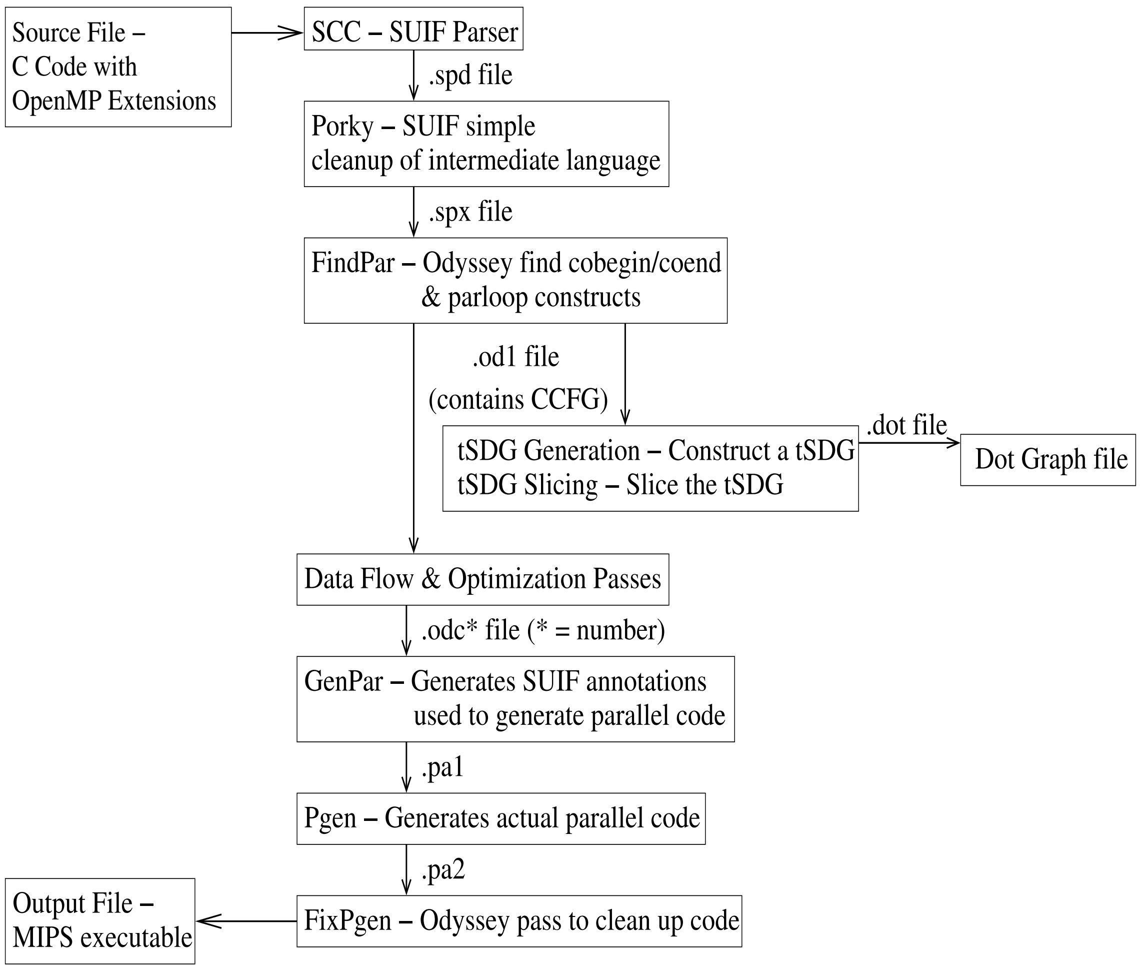

Source File –
C Code with
OpenMP Extensions
SCC – SUIF Parser
.spd file
Porky – SUIF simple
cleanup of intermediate language
.spx file
FindPar – Odyssey find cobegin/coend
& parloop constructs
.od1 file
(contains CCFG)
tSDG Generation – Construct a tSDG
tSDG Slicing – Slice the tSDG
.dot file
Dot Graph file
Data Flow & Optimization Passes
.odc* file (* = number)
GenPar – Generates SUIF annotations
used to generate parallel code
.pa1
Pgen – Generates actual parallel code
.pa2
FixPgen – Odyssey pass to clean up code
Output File –
MIPS executable

**Figure 4.2:** Odyssey and SUIF Passes

# Chapter 5

# RELATED WORK

A great deal of research has already been brought to bear on the topic of slicing. Static sequential slicing was first introduced by Weiser[Wei84]. His techniques for slicing Fortran and Simple-D programs were based on Control Flow Graph representations of program, and did not take into account the calling context of called subroutines. When the Program Dependence Graph representation was developed by Ferrante, Ottenstein, and Warren[FOW87], it was first used for slicing by Ottenstein and Ottenstein[OO84]. Their research determined that slicing based on the PDG representation was a simple graph reachability problem which could be computed in linear time.

The System Dependence Graph representation was introduced by Horowitz, et. al.[HRB97], as an extension of the PDG representation for representing whole programs. The SDG is a system of linked PDG graphs, and allows interprocedural slices to be computed. To solve the calling context problem, Horowitz, et. al. utilizes an attribute grammar and GMOD & GREF sets. Lividas and Croll [LC94] give an alternate algorithm for computation of interprocedural slices. Their techniques build the SDG from the bottom-up by descending into called procedures to process them first. When a called procedure is terminal or has already been solved, the summary information, including transitive dependences, is copied back to each procedure that calls it. Other techniques have been developed to handle programs with recursion[HDC88], and/or to increase precision and safety[SHR91]. A survey of slicing techniques, both static and dynamic, for imperative programs has been written by Tip[Tip95].

The PDG has been extended by various authors to suit their specialized purposes[Che97, DGS92, Kri98, Sar98]. Of these extensions, the research by Cheng [Che97] and Krinke[Kri98] are usable schemas for static slicing of concurrent programs.

Cheng constructs a representation called the Process Dependence Net (PDN) which is later extended to a System Dependence Net (SDN)[Che97]. A PDN represents either the `main` procedure, a free standing procedure, or a method in a class of the program. Additional edges are added to represent direct dependences between a call and the called procedure/method and transitive interprocedural dependences. The SDN is constructed to represent object-oriented features in addition to addressing concurrency issues that arise in concurrent object-oriented programs. Once an SDN has been created for a program, slices of the program can be computed using a simple vertex reachability algorithm[ZCU96].

Krinke introduced the concept of a threaded Program Dependence Graph, which is the basis for his intraprocedural slicing algorithm. Krinke's approach takes into account that dependence between parallel executed statements is not transitive and therefore produces more accurate slices than would be generated by ignoring this fact. Krinke's techniques were used as a basis for developing the techniques for interprocedural slicing of OpenMP programs.

# Chapter 6

# CONCLUSIONS AND FUTURE WORK

This thesis developed the tSDG representation to facilitate the static intraprocedural slicing of OpenMP shared memory parallel programs. OpenMP constructs for worksharing and parallel execution are represented in the CCFG, which forms the basis for the tSDG. Transitive dependencies were refined in the tSDG representation to include the effects of interference dependencies that can occur in shared memory parallel programs. The inclusion of interference edges in our definition of intra-slice paths for computing transitive dependencies (i.e., part of the summary information) at a call-site allowed us to capture the shared memory parallelism dependences in the tSDG. An algorithm for constructing the tSDG was presented. The graph can be sliced interprocedurally by combining interprocedural slicing for sequential programs[LC94] and intraprocedural slicing for parallel programs[Kri98].

The major contribution of this research is the development of an interprocedural slicing technique for shared memory parallel programs, which had not previously existed. OpenMP was targeted because it is the standard for shared memory parallel programs. A slicing algorithm for parallel programs allows the development of tools to shorten debugging time for programmers and to perform automatic program analysis for software maintenance, optimization, and program understanding.

Future work based on this research will expand the range of programs that the tSDG can represent. New program structures to be dealt with include recursion, pointer aliasing, and unstructered control flow. To facilitate tSDG construction on programs with recursion, Lividas and Croll's[LC94] extended call sequence graph will be used. Also, the slicing algorithm will be used in the analysis of hybrid memory consistency models for explicitly parallel programs.

# BIBLIOGRAPHY

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffery D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison Wesley Publishing Company, 1986.

[Che97] J. Cheng. Dependence analysis of parallel and distributed programs and its applications. In *International Conference on Advances in Parallel and Distributed Computing*, 1997.

[DGS92] E. Duesterwald, R. Gupta, and M. Soffa. Distributed slicing and partial re-execution for distributed programs. *5th Workshop on Languages and Compilers for Parallel Computing*, 1992. LNCS 757.

[FOW87] J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. *ACM Transactions of Programming Languages and Systems*, 9(3):319–349, July 1987.

[Gro94] Stanford SUIF Compiler Group. The suif parallelizing compiler guide, 1994. Version 1.0.

[HDC88] J. Hwang, M. Du, and C. Chou. Finding program slices for recursive programs. In *Proceedings of the IEEE COMPSAC 88, IEEE Computer Society*, 1988.

[HRB97] S. Horowitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions of Programming Languages and Systems*, 1997.

[Kri98] J. Krinke. Static slicing of threaded programs. In *ACM SIGPLAN Workshop on Program Analysis for Software Tools and Engineering*, Montreal, CA, June 1998. ACM.

[LC94] P. Livadas and S. Croll. A new algorithm for the calculation of transitive dependences. *Journal of Software Maintenance*, 6:100–127, 1994.

[Lee99] J. Lee. *Compilation Techniques for Explicitly Parallel Programs.* PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1999.

[LMP98] J. Lee, S. Midkiff, and D. A. Padua. Concurrent static single assignment form and constant propagation for explicitly parallel programs. *Lecture Notes in Computer Science*, 1366:114+, 1998.

[MP90] S. Midkiff and D. Padua. Issues in the optimization of parallel programs. In *Proc. Int. Conf. On Parallel Processing*, volume II, pages 105–113, 1990.

[Muc97] Steven S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann Publishers, San Francisco, CA, 1997.

[Nov00] D. Novillo. *Analysis and Optimization of Explicitly Parallel Programs.* PhD thesis, Department of Computing Science, University of Alberta, Edmonton, Canada, 2000.

[NUS98] Diego Novillo, Ronald C. Unrau, and Jonathan Schaeffer. Analysis and optimization of explicitly parallel programs. Technical Report 98-11, Department of Computer Science, University of Alberta, Edmonton, Alberta, Canada, August 1998.

[OO84] K. Ottenstein and L. Ottenstein. The program dependence graph in a software development environment. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, May 1984. ACM SIGPLAN Notices 19,5.

[Ope97] OpenMP Standard Board. *OpenMP Fortran Application Program Interface*, version 1.0 edition, October 1997. http://www.openmp.org.

[Sar98] V. Sarkar. Analysis and optimization of explicitly parallel programs using the parallel program graph representation. *Lecture Notes in Computer Science*, 1366:94–113, 1998.

[SHR91] S. Sinha, M.J. Harrold, and G. Rothermel. System dependence-graph-based slicing of programs with arbitrary interprocedural control flow. In *Proceedings of the 21st International Conference on Software Engineering*, pages 432–441, May 1991.

[SHW93] H. Srinivasan, J. Hook, and M. Wolfe. Static single assignment for explicitly parallel programs. In *Proceedings of the 20th ACM Symposium on the Principles of Programming Languages (POPL)*, pages 260–272, January 1993.

[SOHL$^+$98] M. Snir, S. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra. *MPI: The Complete Reference.* MIT Press, Cambridge, Mass., 1998.

[Tip95] F. Tip. A survey of program slicing techiques. *Journal of Programming Languages*, 3(3):121–189, September 1995.

[Wei84] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10:352–357, 1984.

[ZCU96] J. Zhao, J. Cheng, and K. Ushijima. Static slicing of concurrent object-oriented programs. In *Proceedings of the 20th IEEE Annual International Computer Software and Applications Conference*, 1996.